2003

An object-oriented analysis technique for developing object-oriented simulations in Silk.

Lia. Zannier
University of Windsor

AN OBJECT ORIENTED ANALYSIS TECHNIQUE FOR DEVELOPING OBJECT ORIENTED SIMULATIONS IN SILK

by Lia Zannier

A Thesis submitted to the
Faculty of Graduate Studies and Research
through Industrial and Manufacturing Systems
Engineering in partial fulfillment
of the requirements for the
Degree of Master of Applied Science
at the University of Windsor

Windsor, Ontario, Canada

Your file Votre référence
ISBN: 0-612-84556-7
Our file Notre référence
ISBN: 0-612-84556-7

Abstract

Simulation is very popular as an analysis tool for complex systems. A new paradigm shift towards object oriented simulations is hoping to make simulations even more powerful and accessible. However, in order for those not familiar with object oriented techniques to join this shift, methods must be developed for them to build object oriented simulations, until they acquire an understanding of this programming paradigm. This is an attempt to develop a procedure for just such novice model builders. This thesis begins with some background on object oriented paradigm concepts. It then provides an overview of several different object oriented analysis techniques and attempts to define the common steps of these approaches. Several different commercial object-oriented simulation software are described, with an in depth description of the software selected, Silk, a java based simulation software. The general object oriented analysis approach is then adapted for simulation purposes, resulting in a proposed object oriented analysis technique for developing simulations, and an example is given. Object oriented analysis techniques such as use-case modeling and noun extraction are used to identify potential objects in a discrete manufacturing system to be simulated. These objects are then mapped to existing object classes in the Silk software for final model building.

Dedication

This is dedicated to my parents, Margaret and Danilo, for their encouragement, love and support, and to my sister Carmen, for all of her assistance, both on-topic and off. I guess it's Friday now.

Acknowledgments

I would like to take this opportunity to thank my supervisor, Dr. R.S. Lashkari for his guidance and support in this endeavour. I would also like to thank the committee members, for their suggestions for improvements. I extend my gratitude to Ms. Jacquie Mummery for her assistance, smiling face, and friendly ear. I would like to acknowledge and thank NSERC for their support through the Postgraduate Scholarship program.

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Background

Since its introduction in the 1970s, simulation, especially discrete-event simulation, has become the most frequently used technology in the design and analysis of complex systems [1]. For the number of applications of simulation, there are a seemingly similar number of definitions. Some definitions from literature are listed below:

- Simulation is the imitation of the operation of a real-world process or system over time [2].
- Simulation is an activity whereby one can draw conclusions about the behaviour of a given system by studying the behaviour of a corresponding model whose cause-and-effect relationships are the same as those of the original [3].
- Simulation involves the generation of an artificial history of a system, and the observation of that artificial history to draw inferences concerning the operating characteristics of the real system [3].
- Simulation uses a computer to evaluate a model numerically over a time period and data is gathered to estimate the desired true characteristics of the model [4].

Simulation models use distributions of popular behaviour to account for the randomness of processes. They explicitly model interaction effects, allowing the analyst to measure the impact of process interactions, and generate a variety of statistics to evaluate a system's performance [5].

1.1.1 OBJECT-ORIENTED SIMULATION

A new paradigm in simulation modeling is emerging. This paradigm applies object-oriented techniques and concepts to simulation [6,7,8,9,10]. The appeal of object-oriented simulation lies in its intuitiveness because it is very easy to view the real world as being composed of objects such as machines, workers, parts, etc. Collections of objects, called classes, encapsulate the characteristics and functionality of production system objects. The results of such a paradigm shift will be simulations that are more flexible and reusable. Object-oriented simulation is more flexible in that it addresses the extensibility limitations of simulation by allowing data and procedural abstraction. This differs from traditional languages in which any written procedure cannot use or change the behaviour of pre-existing program structures. Data abstraction allows new data types with their own behaviour to be added arbitrarily to the programming language and interact with existing structures. Another advantage of object-oriented simulation is the ability to preserve most of the developed code for general use in model building. The object definitions are independent of the functions of the system being modeled. This allows for greater reusability, eventually leading to the development of generic object libraries, which will form the foundation of simulation software. A simulation would then be modeled quickly from these generic objects and classes.

A disadvantage of object-oriented simulation is the time required to build the extensive generic object library. True productivity with object-oriented simulation will only be realized once these modeling building blocks exist (as in an existing software). Another disadvantage is the slow processing times and extensive computer resource requirements associated with the programming languages. While in computer science theory, resource requirements are

always an issue, in the field of simulation application, with the ever-increasing accessibility to powerful computers, this may not be as much of a concern.

1.2 Problem Statement

It has been noted in [11] that companies need an easy start when employing a new programming paradigm, as they cannot afford to spend time and resources studying new concepts or constantly hiring consultants. When using object-oriented systems for the first time, they need a simple and effective method. The first stage of this work explores how object oriented principles could be used to create a procedure for developing object-oriented simulations using existing software.

In the literature and in practice, it has been said that identifying and describing the objects in a given system is one of the most difficult tasks in the object-oriented approach, even for those with experience. Therefore, more specifically, the goal of this work is to determine how to identify the objects of a given system to be simulated, for the purposes of building an object oriented simulation. The final goal is one of developing a procedure for inexperienced modelers so they can take advantage of the benefits offered by object-oriented simulation.

1.3 Outline of Thesis

In Chapter 2, a background of object oriented programming concepts is presented, followed, in Chapter 3, by a more in-depth look at object-oriented analysis. Chapter 4 is a discussion of some commercial object-oriented simulation software. A proposed object oriented analysis technique, adapted for simulation, is then described in Chapter 5, followed by an example case in Chapter 6. Both the technique and the example are developed and implemented in the context of manufacturing systems in general. Future work and conclusions are presented in Chapter 7.

Chapter 2: The Object-Oriented Paradigm

2.1 Structured Programming vs. Object Oriented Programming

The basic equation in developing programs is "program = data + control", and different programming paradigms interpret this equation in different ways. There are two types of programming paradigms: data-centric and control-centric. Traditional programming languages follow the control-centric paradigm wherein data flows in and out of small modules that are connected together to achieve a larger task. These structured programming languages have centralized control. Generally a "main" function controls the sequencing and manages all external and global data/information. Object-oriented programming (OOP) follows a data-centric model wherein the building blocks (or modules) of the program are data objects and control is implemented as services that the objects promise to provide to clients (other objects). As no centralized control exists in OOP, there is no "main" function. That is, there is no single object that controls or sequences the flow. Ideally, all objects are created equally and exist at the same level of control. Control is implemented as an interaction between objects that collaborate to achieve functionality.

If one looks at the software lifecycle, the phases that differentiate object oriented development procedures from traditional development procedures are the object-oriented analysis (OOA) and object-oriented design (OOD) phases. For example, in the software life cycle [12], the traditional structured approach has the following stages:

1. Requirements phase
2. Specification phase
3. Design phase
4. Implementation phase

5. Integration phase

6. Maintenance phase

7. Retirement phase

In the object-oriented approach, phases 2, 3, and 4 are replaced with the Object-Oriented Analysis phase, Object-Oriented Design phase, and the Object-Oriented Programming phase, respectively. That is, the difference between structured and object-oriented programming lies in the approaches to analyzing, designing and building the software, as opposed to understanding the requirements, implementing, and maintaining the system.

Similarly in simulation, the difference between traditional approaches and object-oriented approaches is that the entities of the system are considered objects and the model is constructed in terms of objects. That is, the elements of the system being modeled become objects and object classes. Therefore, the difference in simulation procedures would lie in object identification and definition as opposed to traditional programming and logic. Given a basic simulation procedure [13] of:

1. Identifying objectives and constraints
2. Gathering and analyzing data
3. Building the Model
4. Performing Experiments
5. Presenting Results

the incorporation of object-oriented techniques would take place in the "Building the Model" step of the procedure. Other phases would not necessarily be affected. This provides a context for later discussions.

2.2 Basic Object-Oriented Concepts

This section will describe some basic object-oriented programming concepts. In order to support object-oriented programming, a language must have four components: information hiding, data abstraction, dynamic binding, and inheritance [14]. These will be described later. In essence object orientation is a strategy for distributing intelligence across a system so as to maximize cohesion and minimize coupling [15].

The first part of this strategy, cohesion, refers to the degree of interaction of the functions within a module, or more precisely, the extent to which the actions performed by a module are functionally related. A module refers to a block of code that can be invoked in a manner similar to the invocation of a procedure, function, or method. Several different categories of cohesion have been identified and ranked from a low level of cohesion (which is considered bad) to a high level of cohesion (considered good) [12]. The lowest level of cohesion is *coincidental cohesion.* Coincidental cohesion occurs when a module performs multiple, completely unrelated, actions. These unrelated actions co-exist in the same module and achieve their cohesiveness by merely residing in the same block of code. An example would be a module in a word processor that changes the font and prints the document. Aside from existing in the same module, these actions have nothing in common. Coincidental cohesion has two major drawbacks. These modules degrade the maintainability of the product and tend not to be reusable.

The next level of cohesion is *logical cohesion.* Logical cohesion occurs when a module performs a series of related actions, one of which is selected by the calling module. In the case of an elevator, for example, a person is required to press a button to begin the series of tasks for the elevator to arrive at a specific location. Logical cohesion would have the

module contain the action of the person pressing the button as an input when the person has no other relation to elevator control. A drawback here is that the code for more than one action may be intertwined, leading to maintainability problems and difficulty in reuse.

The third level of cohesion is *temporal cohesion,* which refers to a module that contains a series of actions related in time. Typically the actions in these types of modules occur at the same time, but are only weakly related to each other and are more strongly related to actions in other modules. It is difficult to determine and manage how changes in these types of modules affect other modules that may contain related activities. They also tend to be difficult to reuse.

The fourth level of cohesion, *procedural cohesion,* refers to a series of actions related by the sequence of steps to be followed by the product. An example would be a module that reads a customer's identification number and updates the customer's service history. Again, the drawback is that the actions are weakly connected and difficult to reuse in another application.

Communicational cohesion, the next level of cohesion, refers to a module that performs a series of actions related by the sequence of steps to be followed by the product (like procedural cohesion) and performs all the actions on the same data. It improves on procedural cohesion by performing the actions on the same data. In the previous example, the module read the customer's identification number, which is one piece of data, but updated the customer's record with, for example, the date and type of service, which is different data. In communicational cohesion, the module would update the date or type of service in the customer's service history, and send that information to an invoicing program to be placed on the invoice. These two actions are performed on the same pieces of data (i.e.

the date and type of service). While these actions are more closely related than in previously discussed types of cohesion, they still present reusability issues. The goal is to break such a module into separate modules that perform one action.

The next two levels of cohesion are considered to be the highest and, therefore, the best. *Informational cohesion* requires that a module that performs a number of actions has for each of these actions single entry and exit points, and independent code. These actions must all be performed on the same data structure. Each action is completely independent whereas in logical cohesion, they are intertwined. A *functionally cohesive* module is one that performs exactly one action or achieves a single goal. This allows for substantial reuse and maintainability since the module is focused on one task. Informational and functional cohesion are considered to be the highest level of cohesion because the actions are strongly related in their function, yet independent in how they achieve that function. They allow for the greatest reusability and maintainability.

The second part of the strategy, coupling, refers to the degree of interaction between two modules. Coupling is important in that the level of coupling between two modules determines how strongly changes in one module will affect the second module. Again, there are several different levels of coupling that can be ranked from bad (high level of coupling) to good (low level of coupling).

The highest level of coupling, *content coupling,* occurs if one module directly references the contents of the other. An example would include module A modifying a statement in module B. The disadvantage with this type of coupling is not only that any change in A almost always results in a change in B, but also that module A can never be included in another application without including module B.

The next level of coupling, *common coupling*, occurs if two modules have access to the same global data. A common example is two modules having the ability to read and write the same record in a database. This creates disadvantages in terms of readability, maintainability (if a change is made to a global variable in one module, then that change must be made in all the other modules as well), reusability (because an identical list of global variables must be available in any future applications), and controlling data access which tries not to expose the module to more data than it requires.

The third level of coupling, *control coupling*, occurs if one module passes an element of control to another module. For example, ideally if module A asks module B to complete a task, and module B sends a message back to A indicating that it cannot complete the task, module A would then decide how to react to the information. However, if module B sends a message that it cannot complete the task and informs A of the action to take, then modules A and B would be control coupled. Module B is informing module A as to what action to take, as well as passing information. These modules are therefore not independent, which leads to difficulties with reusability.

The fourth level of coupling, *stamp coupling*, occurs if a data structure, such as a record or array is passed to a module, but the module acts on only some of the individual components of that structure. In the case of a company adding a new invoice to the "Amount Due" section of a customer's account, stamp coupling would occur if the invoicing program sends an array containing the customer's name, telephone number, and amount due, to the module responsible for calculating the amount due. The telephone number is information that is not required for the calculation. Stamp coupling can lead to decreased reusability and concerns with uncontrolled access to data.

The best, and lowest, level of coupling is *data coupling.* Data coupling occurs if every argument passed is either a single argument, or a data structure in which every element is used. This level of coupling is the desired goal. The ultimate goal when maximizing cohesion and minimizing coupling is to construct completely individual and independent modules or objects.

The object-oriented paradigm identifies the entities present in a domain, and uses abstractions of these entities to form the classes that will be the foundation of the system. These entities may be tangible objects such as chairs, cars, or people, or they may be roles, interactions or incidents within a system. The relationships between the entities are represented in the interactions that occur between objects in the problem space [16].

An example will be used to explain some of the basic concepts related to the object-oriented paradigm. The hierarchy shown in Figure 1 illustrates a possible categorization of geometrical shapes.

These shapes may be either two- or three-dimensional. If one examines two-dimensional shapes, they can be broken down into circular shapes, triangular shapes, quadrilateral shapes and n-gon shapes. Circular shapes are those that contain 360 degrees and have no sharp corners. Triangular shapes are three-sided shapes containing 180 degrees. Quadrilateral shapes contain 360 degrees and have four sides. N-gon shapes are composed of more than four sides (e.g. pentagons, hexagons, etc.).

Based on this first level of categorization, these shapes will be explored in further detail. One example of a circular shape is an ellipse. A circle is simply an ellipse whose major and minor axes are of equal length. While all triangles have three sides and 180 degrees, the equilateral, isosceles, and scalene differ in the number of sides and angles of

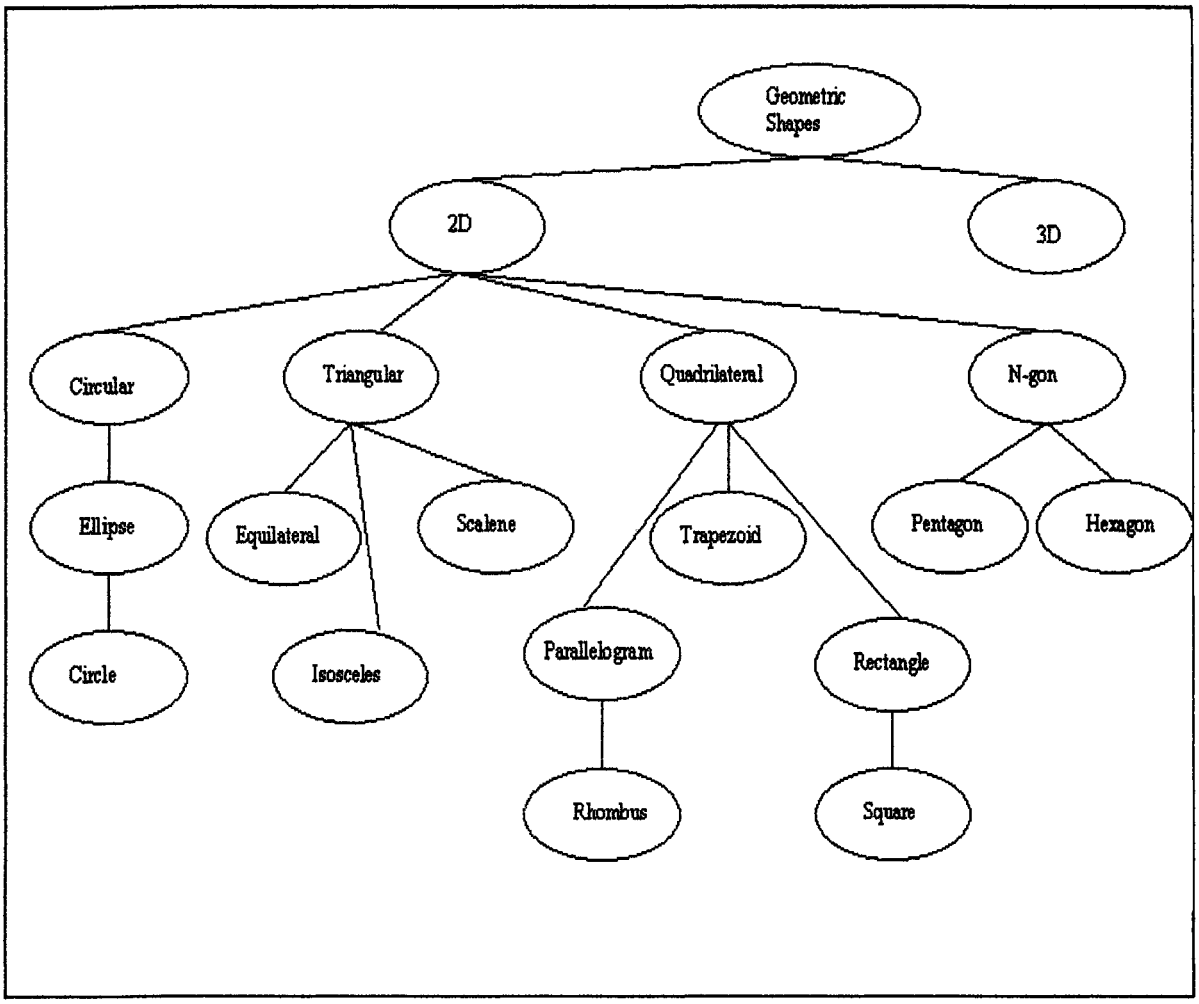

Figure 1: Geometric Shape Hierarchy Example

equal size (three, two and none respectively).

Quadrilaterals can be broken down into parallelograms, rectangles, and trapezoids. A parallelogram is a quadrilateral whose opposite sides must be parallel and equal in length. A rhombus is a parallelogram with the additional constraint of sides of equal length. A rectangle has opposite sides equal in length and all four sides meet at 90 degrees. A square is a special rectangle; its sides must be of equal length. Trapezoids have two parallel sides. N-gons, for the purpose of this example, act as a catch-all to capture any shape with more than four sides. These include pentagons, hexagons, heptagons, octagons, etc. This categorization of shapes will be used as an example in the description of some of the following concepts.

The basis of the object-oriented model is the abstract data type. The abstract data type is a way of representing real-world items, capturing the data representation of the entity, as well as the operations allowed on that data [17]. The idea is to design a data structure with a certain expected behaviour, the details of which are hidden from the users of the structure. The abstract data type also contains the actions to be performed on instances of that data type.

A class is an abstract data type of some concept or entity. The abstraction is meant to capture all the characteristics and behaviours common to all instances of the class. In the shapes example, all the items shown in the figure in the ovals are classes. The quadrilateral class contains all shapes that have four sides and 360 degrees. These fours sides and 360 degrees are the characteristics common to all instances of the class. Similarly, having three sides and 180 degrees is a common characteristics for all triangles. The concept of classes provides a pattern for creating objects and defining the type of object. A class provides all

the information necessary to construct and use its objects. The class is the conceptual modeling tool of the object-oriented programming paradigm. Objects are grouped into classes for specification purposes. A class is a set of possible objects [16]. Objects are instantiated from classes. An important feature of a class is that it supports inheritance (described later) so classes and objects can be arranged in a hierarchy. This hierarchy is easily seen in the shape example in Figure 1. While a class may have multiple instances, each instance generally only has one class (except in the case of multiple inheritance). The methods, or procedures, of the object are stored in the class because all instances of a class contain the same methods [14].

Objects embody an abstraction. They provide services to clients. Clients request the service of the object, and the object executes code, called a method, on the appropriate data to perform the service [18]. Objects are the basic run-time entities in an object-oriented system. That is, while scalene is a class of triangles, the objects are the individual triangles generated during the course of the run of the program. Every object has a set of procedures and functions associated with it that define its operations. The objects are used to model the entities in the application domain [16]. The object's data is represented through its attributes, and its procedures through its methods. An example of an attribute would be the number of sides of the shape, while a method could be to calculate the area. Methods may be classified as public, which are accessible by all objects, or private, which are only accessible to the methods of the class. Public methods simply present a view of the services the class provides, but not the definition of the method [17]. To perform computations, objects send a message to another object to invoke a method. The object receiving the message either

performs the computation based on its methods, or sends a message to another object asking for assistance. Eventually, the result is sent back to the original requesting object. [14]

Inheritance allows for the creation of classes, and objects that are specializations of these classes. The term subclassing refers to creating these specializations. The new class is a subclass of the original class, and the original class is the superclass of the new class. For example, *ellipse* is a subclass of *circular*, while *circular* is the superclass of *ellipse.* Subclasses inherit instance variables, class variables, and methods, but may also add or modify variables and/or methods that are appropriate to these more specialized objects [14]. So while a *rhombus* would inherit the characteristics of having four sides and 360 degrees from *quadrilateral* through *parallelogram,* and the characteristic of having opposite sides parallel from *parallelogram*, it would add the characteristic of having four equal sides.

The inheritance relationship between the classes conveys the specializations and generalizations of the objects represented in the classes. The inheritance relationship allows a class to be defined and implemented based on existing classes. The objects in each class inherit behaviours and actions from the objects and classes above them. This inheritance represents an *is-a* relationship. For example, *square* is-a *rectangle.* Inheritance allows new classes to be created from previous classes.

A similar relationship, the *part-of* relationship, is known as containment and can also be represented in hierarchical form. Given that there is no example in the shapes example, we will use the case of an engine. In the case of *engine* is part-of a *car*, the *car* class contains the engine. The advantage behind these concepts is the ability to write simple and intuitive code.

Another important relationship is the *uses* relationship, more formally referred to as delegation. To perform a service an object might need to "use" the services of another. An object might even promise a service that is provided entirely by another object. This is called delegation. The *uses* relationship is very important and objects can use the services of other objects in various ways. Inheritance is key to constructing software from reusable parts. Say a class *B* inherits from a class *A*. Class *B* is composed of two parts. First, it has a derived part from class *A*, and second it has an incremental part that is specific to class *B*. No matter what additions have been made to class *B*, it will still also be a class *A*. In the case of an *equilateral* triangle, the class derives its three sides and 180 degrees from *triangular.* However, its incremental part constrains the three sides to having equal length.

Inheritance allows for reuse in two ways. First, it encourages the movement of commonalities in objects to the highest level of abstraction possible (that is, as high as possible up the hierarchy). This makes the commonality available to be reused later in the current or a future design. Secondly, it allows for the reuse of an existing class as the basis for the definition of a new class [16]. The concept of multiple inheritance allows a class to have more than one superclass. While this allows for more code sharing, it also creates the possibility of conflicts between multiple superclasses making the system more complex. [14]

Information hiding works under the pretense that users of a class should not be concerned with the internal representation of data members or implementation details of services. If any of these have changed, users of the class should not be notified. Also known as encapsulation, information hiding ensures reliability and modifiability of software systems by reducing the interdependencies among objects [14]. It states that all data within a class should be private and hidden from other objects. Objects are not allowed to know how other

objects are implemented [17]. For example, no other object would know how a square calculated its area. The square would simply report back the area calculated. Encapsulation groups data and the corresponding operations in an object and forms an aggregate. The object's data and procedures are enclosed within a tight boundary, which is impenetrable by other objects. Message passing, another important concept, is a direct result of encapsulation. Because of the impenetrable boundary, for an object to affect the internal condition of another object, the first object must send a message to request the second object to execute one of its procedures. The second object performs the procedure and returns the result to the first. Information hiding also allows maintenance activity to be hidden from the users of the class [16].

Polymorphism is another important object-oriented concept. Achieved through inheritance, it allows the same property to be applied to different objects, or allows users to produce the same behaviour with different objects, depending on the application or circumstance. Essentially, the same command can be given to several different types of objects, with the results being different depending on the object itself. [14]. For example, in the case of the shape example, a circle and a square could receive the polymorphic command *calculate area.* Although the command given is the same, the objects would respond based on their own methods (in this case, different formulae), and the area would be calculated according to different algorithms.

Dynamic binding, one of the necessary components to support object-orientation, refers to the fact that a procedure call is not bound to code until the call is made. Dynamic binding is associated with polymorphism and inheritance in the object-oriented world. That is, a procedure call associated with a polymorphic reference may depend on the dynamic type of

the reference. For example in a graphics program, the procedure *draw* would be defined differently in each geometrical shape. This is another example of a polymorphic reference. Dynamic binding says that the procedure call *draw* will not be tied to any specific code, until the procedure is called. Therefore, *draw* will determine which algorithm to use as soon as the procedure is called [16]. For example, once the procedure is called, *draw* will determine whether the procedure required is one for drawing a square, or a circle.

There are a couple of disadvantages to object-oriented programming. It is more complex to implement object-oriented languages because of the semantic gap between object-oriented languages and typical hardware machines. The programmer must also learn an extensive class library before being proficient in an object-oriented language. However, advantages include shorter development times and a high degree of code sharing. [14]

2.2 Object Oriented Analysis and Design

While analysis concentrates on objects and functionality, which are relevant to the end user, design deals with objects and functions that will be programmed [11]. In object-oriented analysis, the user attempts to model the world by discovering the classes and objects that form the vocabulary of the problem domain. The process of discovering the classes is not easy and involves discovery from domain, invention and elimination. Discovery from domain involves studying the problem domain in terms of its components (machines, people, etc.) to identify objects and classes. Invention refers to the development of objects that may be used for a control purpose. They don't directly relate to entities in the problem domain but will be necessary to carry out the solution. Elimination involves the removal of previously identified objects that turn out to be irrelevant. The result of object-oriented analysis would be the identification of the various classes that are pertinent to the system, as

in the shape example. It is often an iterative process and the result can vary depending on the application. For example, in a different situation, *rectangle* from the shape example could be considered a specialization of *parallelogram*. A rectangle would simply be a special parallelogram that has 90-degree angles. Very often, the hierarchy or model must be modified several times as understanding of the system develops.

Object-oriented design involves designing the software in terms of objects. It can involve detailing modules (classes or objects), selecting algorithms or choosing data structures. Essentially it involves building the software by combining the objects discovered during analysis based on their relationships. In the shape example, object-oriented design would involve translating the hierarchy into code, choosing the appropriate formulas for calculating area and perimeter, etc.

Given that the work presented here will use an existing piece of software, and any existing object-oriented simulation software has already undergone object-oriented design, there is no reason to further examine object-oriented design in the context of this thesis. The purpose here is to identify the objects in a given manufacturing system, in order to better understand how to simulate that system in an existing object-oriented software. Therefore, only object-oriented analysis will be examined from this point on.

Chapter 3: Object Oriented Analysis

The concepts presented in this chapter are discussed from a software development point of view. That is, all of the ideas and tasks described are performed with the final goal of developing a working piece of software. The development of the software is assumed to be in response to a request. As such, it is assumed there is a specific domain that the software will serve and to which it will refer, specific tasks it will be required to fulfill and specific output to be displayed. Any reference to a product or system, in this context, refers to the software itself. Any mention of an analyst refers to a single person or a team of domain experts and software developers. Any reference to inputs, outputs, tasks, uses or knowledge would be information obtained through discussions with the customer or user. A later chapter will determine the relevance to simulation.

3.1 Basics of Object Oriented Analysis

Object oriented analysis is a semi-formal specification technique. The main objective is to identify the classes and objects in a system. Although there are many (over 50 [12]) different techniques for performing object-oriented analysis, they are largely equivalent [12]. There are three basic steps in object-oriented analysis.

3.1.1 USE CASE MODELING

Sometimes referred to as functional modeling, this step involves describing the functionality of the product. The purpose is to provide a general description of the functionality through the identification of various scenarios. Scenarios describe possible events that might take place during the use of the product, thereby encouraging thought about current problems, possible occurrences, assumptions, and risks [19]. This step is concerned

with identifying the interactions of the classes and the user of the system. The intended result is a set of possible situations the system must handle. Scenarios should include typical tasks the user requires the system to perform, but not error states that may be encountered. A use case should describe the behaviour of the system, the event it responds to, the input required, and the possible outputs [20].

3.1.2 CLASS MODELING

Class modeling involves extracting the classes and their attributes and representing them using some type of graphical notation. The class extraction can be performed either by studying the use cases, through noun extraction, or using CRC (Class, Responsibility, Collaboration) Cards. Noun extraction is a three-stage process wherein the system is described in sentences; the nouns of the sentences are taken as candidate classes, and this list of candidate classes is refined. CRC cards are cards filled in to show the name of the class, its responsibility (i.e. its functionality), and the other classes it invokes to achieve its responsibilities (its collaborations). CRC cards can be and have been extended to include the attributes and methods of the class, and are further described in a later section.

3.1.3 DYNAMIC MODELING

The purpose of dynamic modeling is to determine the actions performed by and on each class. The result is generally a state diagram which describes the various possible states a class can attain.

Because object-oriented analysis is a semiformal approach it relies heavily on graphical notation. However, each technique has its own graphical notation. The basic steps of the object-oriented analysis approach are carried out iteratively with changes in one diagram

effecting changes in another. As was mentioned earlier, some of the candidate classes may be eliminated with others added as the steps are carried out.

3.2 Overview of Object Oriented Analysis Techniques

As previously mentioned, there are a large number of object-oriented analysis techniques. Being that they are fairly equivalent [12], it is not necessary, or feasible to describe them all here. This section will describe some of the original and influential approaches, which continue to be popular [21,22]. In keeping with the aforementioned focus on object-oriented analysis, the portions of the techniques that deal with object-oriented design will be omitted. Also, any steps that simply add information to the OOA diagram have been omitted because it is assumed that at some point all of the information discovered will be added to an OOA diagram.

3.2.1 OOA/OOD METHOD BY COAD & YOURDON

The Coad & Yourdon [23] approach is classified as five major activities. The activities are intended to guide the analyst from a high level of abstraction to increasingly lower levels of abstraction. The order of the activities represents the common overall approach. These five activities are described below.

1. Finding Classes & Objects

This activity involves studying the problem domain to find classes and objects. The classes and objects are named using a singular noun or adjective and noun. These names should be taken from standard vocabulary for the problem domain. Objects can be found by observing the system first-hand, listening actively, checking previous OOA results in the same or similar problem domains, checking other systems in the same domain, reading a requesting document from the client, or other references, and prototyping. Potential classes

and objects can be found in many facets of the problem domain. These facets include structures (described in the next activity), other systems or devices the problem domain may interact with, things or events remembered, any roles humans play in the system, operational procedures, physical or geographical sites, and organizational units. The candidate classes and objects are then challenged against several criteria. The criteria names are features the proposed classes should possess. *Needed remembrance* refers to the requirement that the system needs to remember something about the objects in the class. The object should be describable and knowledge of the object should be of interest to the system under consideration. If the system does not need to remember anything about the object, its relevance and validity can be questioned. As long as needed remembrance applies, then the second criterion, *needed behaviour* will follow, as services will be required if for no other reason than to create, connect, access and release objects. The third criterion, *(usually) multiple attributes,* assists in filtering classes when an analyst becomes too low-level in his/her thinking. For example, if an object has only one attribute, that attribute may be better distributed in a number of classes than in an object of its own. Likewise, *(usually) more than one Object in a Class* says that if a Class has just one Object, it should be challenged. If another class with the same or similar attributes and services exists, and if the reality of the problem domain is accurately reflected, the analyst should consider using just one class instead, or using some kind of structure. However, if the reality of the problem domain is reflected by a class with just one object, the analyst may opt to keep the class. *Always-applicable attributes* ensure that every time a system encounters an instance of an object, the instance always has a value for the attribute. The same is true for *always-applicable services*, which checks that behaviour applies to each object in a class. *Domain-based*

requirements are those requirements the system must have (e.g. radars or sensors). Finally, *not merely derived results* avoids results that can be derived such as the age of a client if the birth date is known to the system. Derived results simply clutter the analysis.

2. Identifying Structures

This step involves identifying Generalization-Specification structures, Whole-Part structures and multiple structures. Coad and Yourdon [23] describe a structure as:

> an expression of problem-domain complexity, pertinent to the system's responsibilities. The term "Structure" is used as an overall term, describing both Generalization-Specification and Whole-Part Structures.

A generalization-specification structure can be thought of as an "is a" relationship (e.g. a Car is a Vehicle), while a whole-part structure can be thought of as a "has a" structure (e.g. a Car has an Engine). The notations for the two structures can be seen in Figure 2.

Respectively, the generalization and whole classes are at the top and the specialization and part classes are below, with lines drawn between them. A semi-circle marking indicates a generalization-specification relationship, while a whole-part relationship is indicated by a triangle (which is directional). The ends of the lines in the whole-part structure are marked with amounts or ranges, indicating the number of parts that a whole may have, or the number of wholes that a part may have, at any given moment in time.

When investigating generalization-specification structures for inclusion in the analysis model, each class is considered as a generalization. For its potential specializations, the following questions are asked:

- Is the specialization in the problem domain?
- Is it within the system's responsibilities?

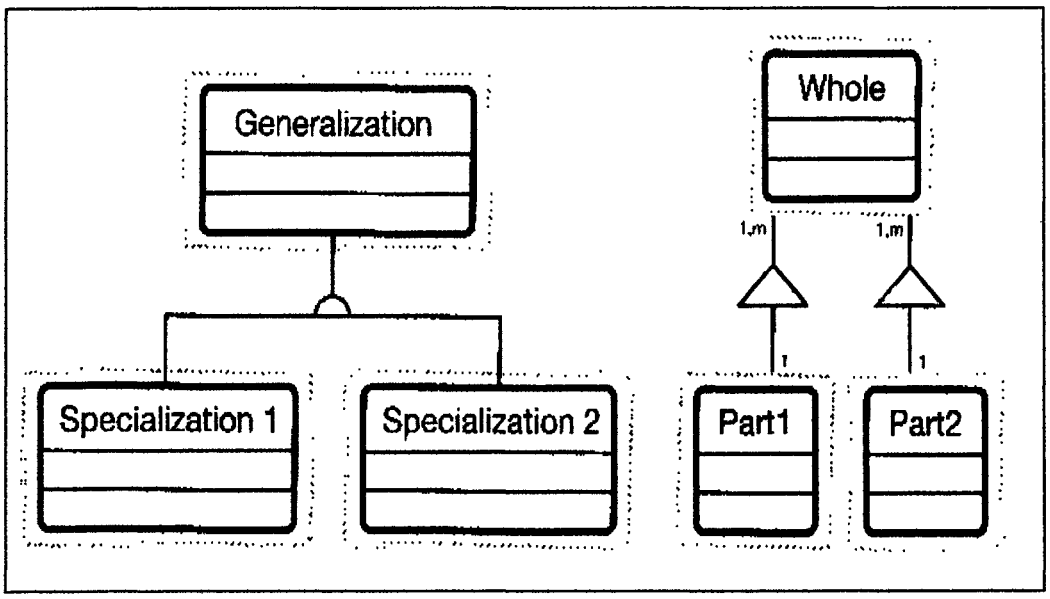

Figure 2: The Coad & Yourdon representation of generalization/specialization and whole/part relationships (taken from [23])

- Will there be inheritance?
- Will the specializations meet the "what to consider and challenge" criteria in the first activity?

The same class is then considered as a specialization, and, for its potential generalizations, the same questions are asked. Structures can also be identified by checking previous OOA results in the same and similar problem domains. The most common form of the generalization-specification structure is the hierarchy, although lattices (also known as multiple inheritance) can also be used to highlight additional specializations or capture commonality.

When considering whole-part structures, the analyst should, in addition to checking previous OOA results, consider variations such as assembly-parts, container-contents, and collection-members. Then, like before, while considering each object (not class) as a whole, for its potential parts, ask the questions:

- Is it in the problem domain?
- Is it within the system's responsibilities?
- Does it capture more than just a status value? (If not, include it as an attribute within the *whole*)
- Does it provide a useful abstraction in dealing with the problem domain?

The same questions are then asked considering the object as a part, for its potential whole. Multiple structures include combinations of generalization-specification structures and whole-part structures.

3. Identifying Subjects

Subjects are mechanisms for guiding model readers through a large, complex model. The subjects act as parts used to communicate the overall problem domain and the system's responsibilities. They essentially compartmentalize classes to allow for easier interpretation of the model. To identify subjects, the uppermost class in each structure is promoted upwards to a subject. Then, each class not in a structure is promoted upwards into a subject. The subjects are then refined by using problem sub-domains, focusing on minimizing interdependencies and interactions.

4. Identifying Attributes

An attribute is a piece of data whose value is specific for each object in a class. It describes a value kept within an object that is to be used exclusively by the services of that object. Examples of attributes would be the colour or engine size of a specific "car" object, or the weight or age of a specific "person" object. To identify the attributes of an object, the following questions should be asked:

- How is the object described in general?
- How is the object described in this problem domain?
- How is the object described in the context of this system's responsibilities?
- What does the object need to know?
- What state information does the object need to remember over time?
- What states can the object be in?

Again, previous OOA results in the same or similar domains can be checked for additional, appropriate attributes. The attributes should then be put in the class and object that they best describe, depending on the problem domain. If the classes are contained within

generalization-specification structures, the more general attributes should be placed higher in the structure and the more specialized attributes lower. Attributes should be placed as high as possible within the structure, while still applying to all of the specializations.

Instance Connections are used to model association between objects. They are models of problem domain mappings that one object needs with other objects, in order to fulfill responsibilities. The constraints between objects can be represented as a single value or as a range. For example, a particular flight plan is assigned to only one aircraft, but a specific aircraft may have zero to many flight plans. For each object, connection lines are added from it to other objects, to reflect mappings within the problem domain and within the system's responsibilities. The lower and upper bounds on the range are then added.

Each attribute should be checked for a value of "not applicable" or repeating values. Classes should be checked to see that they contain more than one attribute. Instance connections should be checked for a many-to-many relationship, multiple connections between objects, additional needed connections, or connections between objects of the same class. To finish, attributes should be named, and a short description recorded. Constraints may also be added.

5. Identifying Service

Services are specific behaviours that an object is responsible for exhibiting. The first step in defining services is to identify the object states. The values of the object's attributes represent the state of the object. To identify the object states, the analyst first looks at the potential values of the attributes, and then determines whether different behaviours are required to obtain those potential values. For example, if an attribute has the potential values "on" and "off", the behaviour required to change that value would be noted.

The second step is to identify the required services. Required services fall into two classifications. The first, *Algorithmically-Simple Services* follow the same basic pattern, over and over, and apply to each class in the model. They may be specified once, and are treated as implicit services (i.e. they are not explicitly shown the diagram). There are four algorithmically-simple services. The service *Create* creates and initializes a new object in a class. The service *Connect* connects an object with another, effectively establishing a mapping between two objects. The service *Access* gets or sets the attribute values of an object. The service *Release* disconnects, deletes, or releases an object. The majority of required object behaviour can be covered using these services. The second classification is *Algorithmically-Complex Services*, which are broken down into *Calculate* and *Monitor*. Like their names suggest, the *Calculate* service calculates a result from the attribute values of an object, while the *Monitor* service monitors an external system or device. These additional services are identified by examining an object in its states and asking:

- What calculations must the object perform on its values?
- What monitoring must the object do to detect and respond to change in an external system or device?

The third step is to identify message connections. Message Connections are mappings from one object to another, where one object, the sender, sends a message to another object, the receiver. To identify the message connections, the following questions are asked of each object:

- From what other objects does it need services?
- What other objects need one of its services?

The fourth step is to specify the services. This is done using a service chart within a Class & Object template. The service chart is like a flowchart of the service broken down into its various components.

The final step is to put the documentation together, including the OOA model, the class and object specifications, and any supplemental documentation needed. An example of the OOA model for a vehicle registration/title system can be found in Figure 3.

3.2.2 Designing OO Software (DOOS) by Wirfs-Brock et al.

Only the initial exploratory phase from this approach will be described. The detailed analysis phase does not apply here. Again, Wirfs-Brock et al [24] describe the goal of OO techniques as attempting to reduce the complexity of the world through abstraction.

1. Identify Classes

The first step in this stage is reading and understanding the requirements specification. If no specification is available, a description of the goals of the design should be written. The analyst then reads the specification or description again, noting all noun phrases. These noun phrases will become part of the candidate list of classes. If the class can be named, or if a statement of purpose can be formulated for the class, the chances are higher that it will be included in the final design. There are also a series of guidelines applied to the noun phrases to determine which are strong contenders for candidate classes. The guidelines are:

- Model physical objects (e.g. printers, disks, etc.).
- Model conceptual entities that form a cohesive abstraction (e.g. a file).
- Choose the word that is more meaningful relative to the rest of the system, if there is more than one word used for a concept.

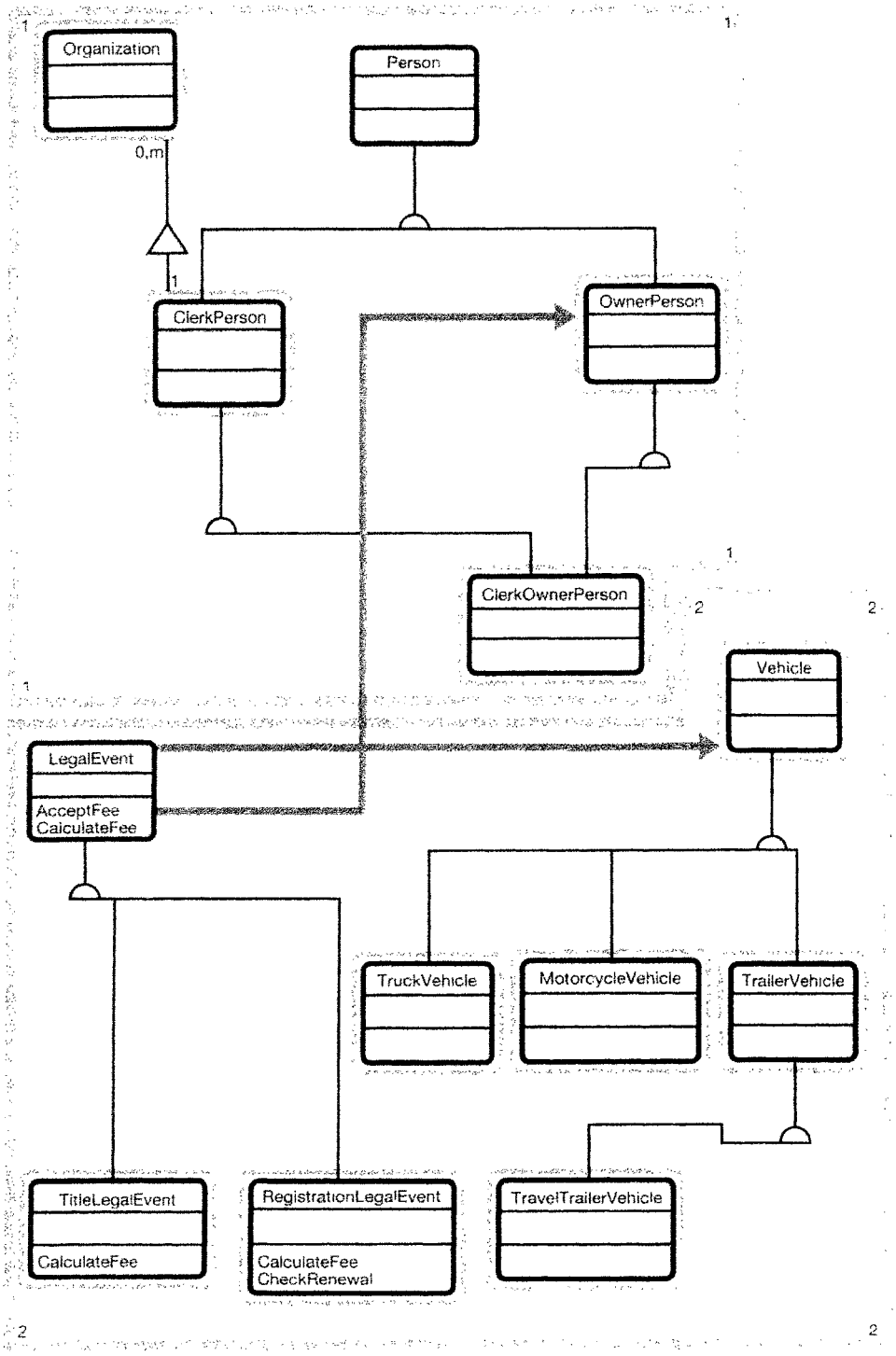

Figure 3: An example of Coad & Yourdon's Object-Oriented Analysis Diagram (taken from [23])

- Be wary of adjectives, as they can suggest a different kind of object, a different use of the object, or be irrelevant.
- Check sentences in the passive voice for hidden subjects that may be candidate classes.
- Model categories of classes.
- Model known interfaces to the outside world (e.g. user interface).
- Model values of attributes of objects, but not the attributes (e.g. float or integer as opposed to length or size).

The result of the noun extraction and the implementation of these guidelines will be the first list of classes for the program.

The second step in this process is to record the candidate classes. This approach suggests using index cards with one class per card. The back of the card should contain a short description of the overall purpose of the class. This description may help to clarify the motivation in creating the class and serve as the basis for class documentation later. Index cards are considered beneficial because of their compact size, ease of manipulation (especially when arranging and organizing them), and ease of use when modifying or discarding them.

The third step is to find abstract classes. Abstract classes will assist in specifying the structure of the software, as well as in identifying missing classes. Abstract classes will be developed from classes that share an attribute or behaviour. If several classes share a behaviour, an abstract superclass should be designed to capture that behaviour in one place. Each subclass would then inherit the behaviour. Candidate abstract superclasses can be found by grouping related classes. Once a group has been identified, the superclass that

represents it can be named, using a singular noun or noun phrase. The categories of classes identified in the guidelines above also serve as the groups. The superclasses, and subclasses are then recorded on the class index cards. The classes can then be examined, in the context of the system, to find missing classes.

2. Identify Responsibilities

Responsibilities include the knowledge an object maintains and the actions an object can perform. They suggest the purpose of an object and its place in the system. They are intended to represent only publicly available services. Responsibilities can be found through the requirements specification, or through the classes. In the requirements specification, responsibilities may be found in verbs or anywhere information that must be maintained and manipulated by an object is mentioned. It is also useful to perform a walk-through of the system, imagining how the system will be used, and running through various scenarios. When looking at the classes, responsibilities can be gleaned from class names, roles the classes are to play within the system, and the statement of purpose of each class.

Each responsibility is then assigned to the classes to which it should logically belong. If there is any question as to which class the responsibility clearly belongs, the following guidelines are suggested:

- Evenly distribute system intelligence (assuring each class is fairly equal in terms of system intelligence)
- State responsibilities as generally as possible.
- Keep behaviour with related information.
- Keep information about one thing in one place.
- Share responsibilities among related objects.

Responsibilities can also be found by examining relationships between classes. Although they may not apply in every situation, there are three particularly useful relationships. The *is-kind-of* relationship is present when one class seems to resemble another class. This is often a sign of a superclass-subclass relationship, and is marked by a shared attribute. This allows the responsibilities to be assigned to the superclass, and inherited by each subclass, thus allowing the responsibility to be assigned only once. The system is then re-examined for "is kind of", "is analogous to" and "is part of" responsibilities. The *is-analogous-to* relationship is present when two separate classes bear an analogous relationship to another part of the system. This relationship is often indicated by the presence of same responsibilities in each class. This type of relationship may indicate a missing superclass to which the shared the responsibilities can be assigned. The *is-part-of* relationship occurs when a class seems to be a part of another class. This relationship in no way implies shared behaviour (e.g. an engine does not have the same behaviour as a car), and so assists in assigning responsibilities by narrowing the scope of possibilities. The relationship also helps in identifying responsibilities related to maintaining information since an object composed of other objects must usually know about its parts. Unassigned, related, responsibilities may be included by adding a new class, or some responsibilities may be arbitrarily assigned to an object. These last two methods must be based on the judgment of the analyst, in the context of the system. The responsibilities are then recorded on the class cards, in the form of a succinct phrase.

3. Identify Collaborations

Collaborations are requests from a client class to a server class in fulfillment of a client responsibility. That is, an object collaborates with another object if it sends the other object a

message to fulfill a responsibility. It is possible that several collaborations are required to fill one responsibility. Through the pattern of collaborations, the flow of control and information within the application is revealed. The identification of collaborations will lead to classifying various objects as clients and servers, which may serve to identify missing or misplaced responsibilities.

Finding collaborations begins by analyzing the interactions of each class and examining the responsibilities for dependencies. The following questions serve as guidelines to identify collaborations.

- Is the class capable of fulfilling this responsibility itself?
- If not, what does it need?
- From what other class can it acquire what it needs?
- What does this class do or know?
- What other classes require the result or information?

Any classes that have no interactions with other classes should be considered for removal. Before discarding classes, several walk-throughs should be performed to ensure that the class has no interactions.

Collaborations can also be found through specific relationships, depending on the application. The *is-part-of* relationship may imply a responsibility for maintaining information, thus indicating a collaboration. The *has-knowledge-of* relationship is present when classes know about other classes. These may indicate a responsibility to know information and thus, a collaboration between the class that has knowledge and the class that is known. The *depend-on* relationship occurs when classes are hooked together, possible changing together. This relationship implies a has-knowledge-of relationship, or the

presence of a third class that forms the connection. Either way, it may result in a collaboration.

The collaborations are recorded on the class card of the client class. To indicate the collaboration, the name of the server class is written directly to the right of the responsibility the collaboration fulfills. This approach then suggests taking several walk-throughs of the system, using various scenarios to check for completeness.

3.2.3 Object Modeling Technique by Rumbaugh et al.

The Object Modeling Technique (OMT) [25] divides analysis and design into three parts: analysis, system design, and object design. The analysis model represents what the desired system must do, as opposed to how it will operate. At this stage, the objects represent application-domain concepts, while implementation concepts such as data structures are ignored. Application experts who are not programmers should easily understand the model. System design involves decision making about the overall architecture and organizing the system into subsystems based on the analysis model and the proposed architecture. The system designer makes tentative resource allocations, decides the performance characteristics to optimize and chooses a strategy for tackling the problem. The design model built during the object design phase is based on the analysis model but contains implementation details. The details focus on the data structures and algorithms needed to implement each class. In implementation, the object classes and relationships developed during the previous phases are converted into a programming language. As most of the difficult decisions are made in the design phase, this stage should be of a more minor, mechanical nature. Only the analysis section will be described in depth here.

The OMT approach describes the system using three different types of models: the object model, the dynamic model and the functional model. These three models form the core of the OMT method, described below.

1. Write a Problem Statement

The analysis model begins by writing a problem statement for the problem domain. This statement of the requirements of the system should state what is to be done, should be a statement of needs, and should indicate which features are mandatory and which are optional. Any attempt to describe a possible solution, or how the system will be developed does not belong in this step. The problem statement may have any amount of detail; in fact, many problem statements are incomplete or ambiguous. The purpose of the analysis stage is to gain a full understanding the problem and all its implications.

2. Build Object Model

The purpose of the object model is to represent the static data structure of the system and organize it into practical pieces. One of the most crucial aspects is the organization of the top level of the system into classes, which are connected by associations. The problem statement and domain experts provide the knowledge for this stage. There are several stages in the building of an object model.

- Identify objects and classes.
- Prepare a data dictionary.
- Identify associations between objects.
- Identify attributes of objects.
- Using inheritance, organize and simplify object classes.
- Verify access paths exist for likely queries.

- Refine the model through iterations.
- Group classes into modules.

The first step in the object model is to identify objects and classes using noun extraction. Objects and classes are identified by looking at the nouns in the problem statement. The candidate list should contain as many objects as possible at this stage. Once the list is complete, objects are discarded if they can be classified as any of one of the following criteria. *Redundant classes* are classes that express the same information. The more descriptive class should be kept. For example, in the case of a retail store, a Customer class should be kept and a User class discarded. *Irrelevant classes* are classes that have little or nothing to do with the problem. This requires some judgment based on the application. *Vague classes* may include classes that have ill-defined boundaries or classes that are too broad in scope. *Attributes* are items that primarily describe individual objects, such as name, address, age, etc. *Operations* are discarded if a name describes an operation that is applied to objects and not itself manipulated. The names of classes should reflect their fundamental nature, and not the *Roles* they play in an association. For example, in a car manufacturer's database, the class Customer would be preferable to Owner, as Owner does not allow for additional drivers or people who have leased the vehicle. *Implementation constructs*, that is, any construct extraneous to the real world, such as subroutines and algorithms, should left to a later design stage.

The second step is to prepare a data dictionary since isolated words are too open to interpretation. A small paragraph describing the scope of the class, as well as any assumptions or restrictions, should be written about each of the remaining classes. The third step is to identify associations between classes. An association is any dependency between

two or more classes, and usually corresponds to verbs and verb phrases in the problem statement. Again, unnecessary and incorrect associations are discarded based on a set of criteria. *Associations between eliminated classes* discards any association that involves one of the classes eliminated in the first step of this process. *Irrelevant or implementation associations* eliminate any associations that are outside the problem domain, or are concerned with implementation constructs. *Actions* are transient events whereas associations should only describe a structural property of the application domain. *Ternary associations*, which are associations between three or more classes, can generally be decomposed into binary associations. However, they may sometimes be required. *Derived associations* are associations that can be defined in terms of other associations because they are redundant. For example, Grandchild Of can be expressed in terms of two Child Of associations.

The next step in object modeling is to identify attributes. These are properties of individual objects and generally correspond to nouns with possessive phrases. They are less likely to be fully described in the problem statement, so domain knowledge must be used to find them. The subsequent step organizes classes by using inheritance to reveal common structure. Inheritance can be added from bottom up by generalizing common aspects of existing classes into a superclass or top down by refining existing classes into specialized subclasses. The process of testing access paths traces various paths through the object model diagram to see if they yield sensible results. This step also allows the analyst to verify that all questions that are likely to be asked of the system will result in a solution. The final step in object modeling is to iterate through the process, making refinements to the object, attributes, and associations, as well as the hierarchy, as needed. An example of an object diagram for an ATM system can be found in Figure 4.

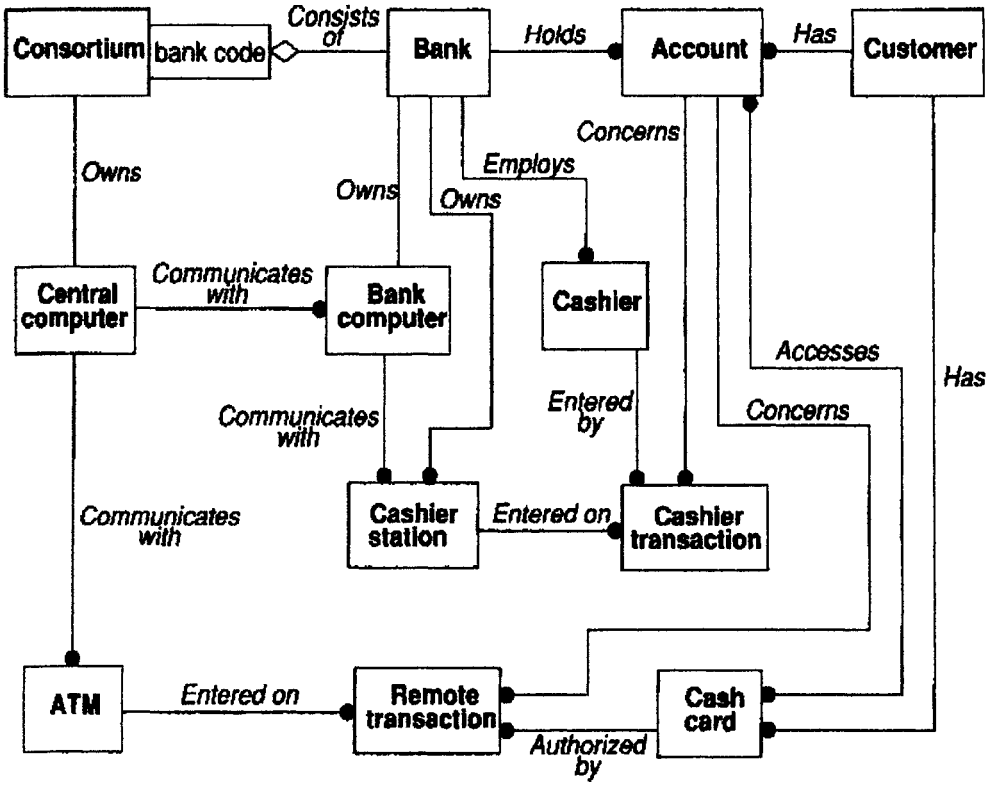

Figure 4: An example of Rumbaugh's object diagram (taken from [25])

3. Build Dynamic Model

The dynamic model reveals the time-dependent behaviour of the system and its objects. It contains the following steps:

- Prepare scenarios of typical interaction sequences.
- Identify events between objects.
- Prepare an event trace for each scenario.
- Build a state diagram.
- Match events between objects to verify consistency.

The first step is to prepare the various scenarios, which show the major interactions and information exchanges. The scenario is a sequence of events, where events occur whenever information is exchanged between an object in the system and an outside agent, such as a user. The parameters of the event are the information values exchanged. For each event, the actor that caused the event and the parameters of the event are identified. Scenarios should be prepared for normal, special and error cases.

In the second step, identifying events, events include all actions to or from users or external devices such as signals, inputs, decisions, and transitions. All events that have the same effect on flow of control (e.g. entering a password) are grouped together under a single name to form classes. Each type of event is then allocated to the object classes that send and retrieve it. Each scenario is shown through an event trace, which is an ordered list of events between different objects. The events among a group of classes are shown on an event flow diagram, shown in Figure 5.

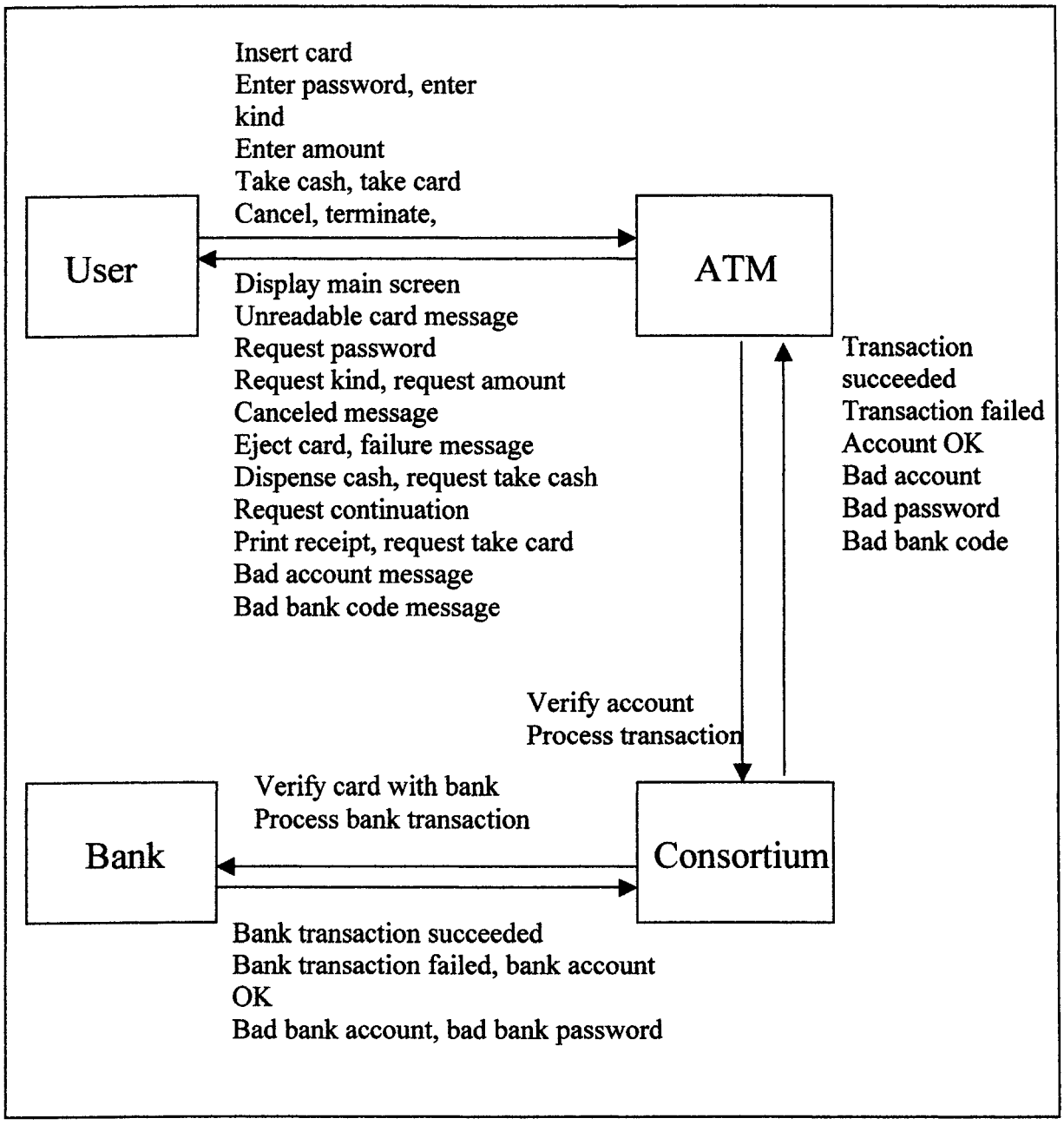

Figure 5: An example of Rumbaugh's event flow diagram (taken from [25])

The third step is to develop the state diagrams, which show the events the object receives and sends. Every scenario or event trace will correspond to a path in the state diagram. The building of the diagram begins with an event trace that affects the class, and is a typical interaction. The events are arranged into arced paths in the diagram. Once the first event trace has been mapped, other scenarios are added to the diagram. After the normal events have been added, the special cases are added. The state diagram is complete when all scenarios are covered in the diagram. Posing "what-if" questions can test completeness. This process is repeated for each class of objects, concentrating on those with important interactions.

The last step in dynamic modeling is to check for completeness at a system level. Most states should have a sender and receiver and input events could be checked from object to object to ensure scenarios have been matched. An example of a state diagram can be seen in Figure 6.

4. Build Functional Model

The functional model shows how values are computed, which values depend on other values, and the functions that relate them. This is represented in the form of a data flow diagram. The construction of a functional model contains the following steps:

- Identify input and output values.
- Build data flow diagrams showing functional dependencies.
- Describe functions.
- Identify constraints.
- Specify optimization criteria

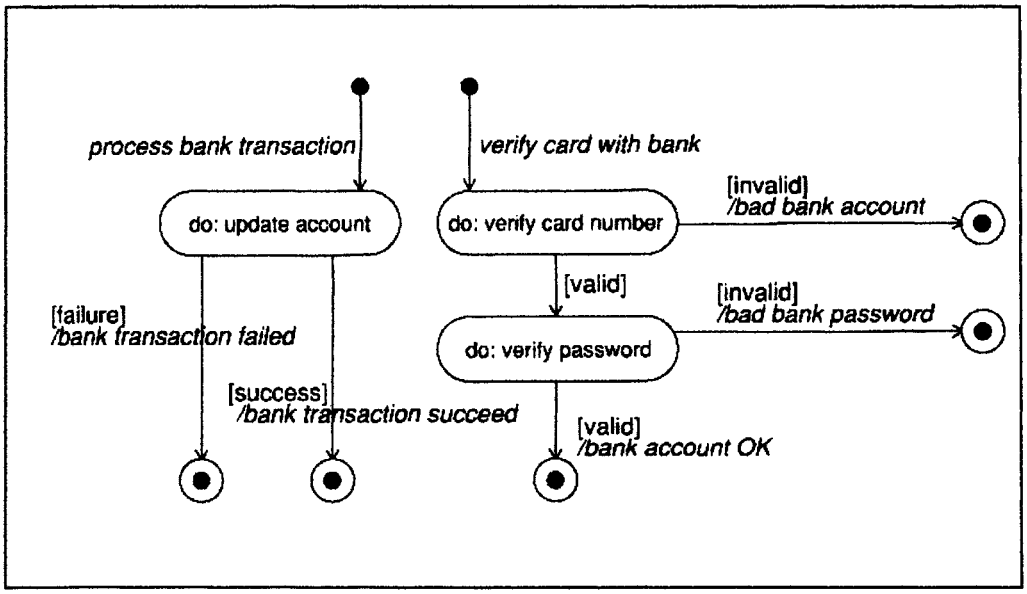

Figure 6: An example of Rumbaugh's event flow diagram (taken from [25])

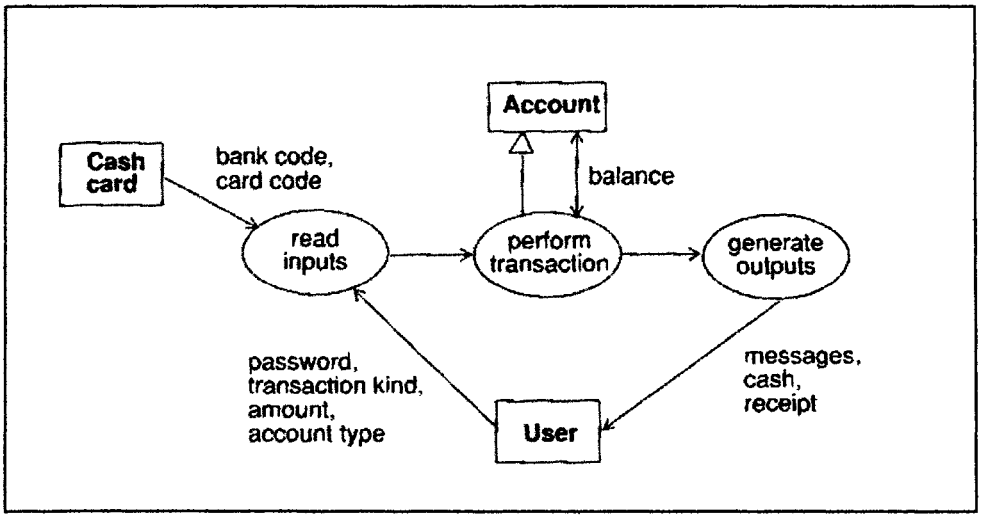

Figure 7: An example of Rumbaugh's data flow diagram (taken from [25])

In the first step, input and output values are the parameters of events between the system and the outside world. An example of an input value would be a password (the actual combination of numbers/letters/symbols), while an output value could be an action verification message. The second step is constructing the data flow diagram, usually consisting of layers, which shows how each output value is computed from input values. Each nontrivial process in a top-level diagram can be expanded into a lower-level diagram. An example of a data flow diagram can be seen in Figure 7.

In the third step, describing functions, the description of the function can be in natural language, mathematical equations, pseudocode or any appropriate form. The description can be declarative, which specifies the relationships between input and output values and the relationships among the output values, or procedural, which specifies a function by giving an algorithm to compute it. It is important to note, at this stage, that the key is to capture **what** the function does, not **how** to implement it. The fourth step is to identify constraints between objects, where constraints are functional dependencies that are not related to input-output dependencies. In a banking system, a constraint could be that no account may have a negative balance. The times and conditions under which the constraints are held are noted here as well. The last step, specifying optimization criteria, involves identifying any values to be maximized, minimized or optimized in some other way.

5. Adding Operations

This approach places much less emphasis on defining operations in the analysis phase. Operations are identified from, and correspond to attributes or associations in the object model, events in the dynamic model, and functions in the functional model.

6. Iterating the Analysis

As with many methodologies, the analysis stages should be iterated to produce a cleaner design and the final model should be verified by the requestor.

3.2.4 OBJECT ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS BY BOOCH

According to Booch, the hardest part of object-oriented analysis and design is the identification of classes and objects. The Booch method is divided into a macro process and a micro process [26]. The macro process consists of conceptualization, analysis, design, evolution, and maintenance. The micro process will be described in more detail because it is the more relevant portion of the technique.

1. Identify Classes and Objects

The purpose of this step is to identify the boundaries of the problem. Through the discovery of the abstractions that will form the vocabulary of the problem domain, the problem becomes constrained by deciding what is and is not of interest. The result of this step is a data dictionary that is updated as the project develops. The data dictionary acts as a repository for the abstractions (e.g. objects, classes, possibly attributes) relevant to the system. It establishes a common vocabulary for the project, as well as serving as an efficient tool with which to browse through all the elements in the system. It allows the architects of the system to obtain a global view of the project, which may lead to the discovery of commonalities. Booch suggests three ways that objects and classes might be identified. The first is through the use of classical approaches to object-oriented analysis. These approaches generate a set of candidate classes and objects by classifying the system into tangible things, the roles they play, events, interactions, organizations, etc. He references the classifications of Coad & Yourdon, and Shlaer & Mellor, described in this section. The second technique

mentioned is the use of behaviour analysis, in which abstractions are related to the system's function points. In this technique, objects and classes are identified through the discovery of common responsibilities and behaviours. A function point is any outwardly visible and testable behaviour of the system relevant to the problem domain. The third technique is use-case analysis, in which end users and domain experts list the scenarios that are fundamental to the system's operation. These will form the functionality of the application. Each scenario is then studied, with the team walking through each scenario to identify the objects that participate in the scenario, their responsibilities, and how they collaborate. Like other approaches, the list of objects and classes identified using these techniques will change as further analysis continues. This first stage is considered complete once a fairly stable data dictionary is formed. The dictionary is considered stable once it doesn't drastically change as the analyst iterates through the process.

2. Identify Semantics of Classes and Objects

The goal in this step is to establish the attributes and behaviour of each class from the previous phase. The three activities associated with this step are storyboarding, isolated class design and pattern scavenging. Storyboarding, a top down approach, begins with a scenario or a set of scenarios related to a single system function point. The abstractions from the previous phase relevant to the scenario are identified. The team performs a walk through of the activity of the scenario, assigning responsibilities to each abstraction. This is repeated for each scenario of system behaviour. As the storyboarding process continues, responsibilities are reallocated to distribute the behaviour in a balanced way. Storyboarding may be illustrated via CRC cards or object diagrams. The semantics may be specified by describing the responsibility using a phrase or single sentence.

Isolated class design, a bottom up approach, focuses on a single abstraction, and considers the operations that will complete that abstraction. For each abstraction, its roles and responsibilities are enumerated. Operations are then developed to implement those responsibilities. Each operation is considered in turn to ensure it is primitive. If it is not, it is broken down to expose the primitive operations. It is important to consider the need for completeness, which may result in operations being added that are not required for immediate purposes, but may round out the abstraction and be used in future cases. However, it is encouraged that the team lean more toward simplicity than complexity.

Pattern scavenging, emphasizes the importance of commonality by suggesting that if the analyst is sensitive to patterns of behaviour, opportunities for reuse will appear. The set of scenarios should be investigated for patterns of interaction among abstractions. The set of responsibilities should be examined for patterns of behaviour. As concrete operations are specified, they should be examined for patterns within operations. Because the semantics of a class encompass its roles, responsibilities, and operations, this phase is considered complete once a reasonably sufficient, primitive, and complete set of responsibilities and/or operations for each abstraction has been discovered.

3. Identify Relationships among Classes and Objects

The identification of relationships among classes allows boundaries to be solidified and collaborators to be recognized earlier in the process. The three activities in this step are: the specification of associations, the identification of various collaborations, and the refinement of associations. Associations are considered semantically weak in that they only represent some sort dependency, the role and cardinality of each participant, and maybe a statement of navigability. The first step in specifying the specifications is to collect a set of classes at a

given level of abstraction, or associated with a particular family of scenarios. The important operations and attributes of each abstraction, relevant to the problem being modeled, are identified. An association is established if there is deemed to be a dependency between any two classes. Examples of an association may be the need to navigate from one object to another, or the need to elicit some behaviour from an object. For each association, the role of each participant, as well as any relevant cardinality or constraint, is identified. Walk-throughs are used to validate the scenarios.

The identification of collaborations is a problem of classification, and is primarily a design activity, as opposed to an analysis activity. It involves placing classes in a generalization/specialization hierarchy, clustering classes into categories, organizing modules into subsystems, and allocating classes and objects to modules. These tasks are generally performed using scenario walk-throughs, identifying commonalities and identifying visible relationships.

The refinement of associations evolves associations into more precise relationships as problem domain understanding increases. This could involve several techniques, such as looking for patterns of behaviour that represent opportunities for specialization/generalization in a given collection of classes. New classes can be added if there are patterns of class structure, or behaviourally similar classes not related in the hierarchy can illustrate the need for additional common classes. This phase is complete once the semantics and relationships among the more interesting abstractions have been specified.

4. Implementation of the Classes and Objects

The only step in this activity is to select the structures and algorithms that provide the semantics of the abstractions (classes and objects) identified earlier. In the analysis portion

of this method, the results of this step are relatively abstract. This activity is interested in identifying the abstractions to which responsibility can be delegated. Objects and classes are considered individually, and central operations and objects, which can be delegated responsibility, are identified. This phase is completed in more details in design when suitable algorithms are selected.

3.2.5 OBJECT LIFECYCLES BY SHLAER AND MELLOR

This approach [27] divides system development into OOA and OOD. Again, the OOD portion will be ignored. The object-oriented analysis portion can be described in three steps.

1. Information Modeling

The information modeling step identifies the objects that make up the system. The information model depicts not only the objects, but also the attributes and relationships of the objects. Objects are identified through the conceptual entities in the problem being analyzed. Though they can be of a physical or abstract nature, they tend to fall into the following categories. *Tangible objects* are abstractions of a thing in the physical world, such as a car or a machine. *Roles* abstract the purpose or assignment of a person, organization, or piece of equipment, such as an employee of a company. An *incident* is an object that abstracts an occurrence or some happening. *Interaction* objects result from associations between other objects. Rules, standards or quality criteria, such as a recipe, are represented through *specification objects*. Each object is then described using a short statement, which will be used to determine whether a real world object is or is not an instance of the system object.

Attributes are abstractions of the characteristics of the objects. Attributes may be determined to be an identifier if the attribute value uniquely distinguishes each instance of an object. For example, a Name attribute would be an identifier, but Weight would not since

two people could have the same weight. An object must have at least one identifier. There are three different types of attributes. *Descriptive attributes* provide fundamental facts of the object, such as the attribute Balance in the object Savings Account. Changes in the value of a descriptive attribute changes only some aspect of the instance, but the instance remains the same (i.e. a change of $5 in a bank balance does not change the bank account itself). *Naming attributes* name or label instances and are frequently used as identifiers. Again, changes in these kinds of attributes do not change the instance itself. *Referential attributes* tie an instance of one object to an instance of another, such as the attribute Customer ID that indicates which customer owns a particular account. If this type of attribute changes, different instances become related. For each attribute, a short description of the characteristics being captured, as well the domain of values for the attribute is described. When identifying and modeling attributes four rules can be applied. The first rule is that one instance of an object has exactly one value for each attribute at any given time. The second rule, that an attribute must contain no internal structure, means that an attribute value cannot be broken down to obtain alternate information. The third rule states that when an object has a compound identifier (i.e. made up of more than one attribute), the attributes that are not part of the identifier represent characteristics of the entire object, and not the characteristic of part of the identifier. For example, if a pipeline had as its identifiers Source Tank and Sink Tank, and had Gallons as a remaining attribute, Gallons would represent the amount transferred between the two tanks, as opposed to the level in either of the tanks. The fourth rule states that each attribute that is not a part of an identifier represents a characteristic of the instance, and not a characteristic of one of the other nonidentifier attributes. For example, if a batch of parts to be heat-treated has as an identifier Batch ID, and has as other attributes

Plan ID and Cooking Time, the Cooking Time attribute would represent the actual time the batch was cooked, and not the cooking time specified in the Plan ID.

The relationships between objects are then modeled and described. Relationships are abstractions of the associations between different kinds of things in the real world. These real world things are represented as objects. Relationships may be unconditional, meaning that every instance of both objects participates in the relationship, or conditional, meaning that some instances of the objects may not participate in the relationship. The relationships must be described using an identifier, the names of the relationship from the point of view of both objects (e.g. owns and is-owned-by), the multiplicity and conditionality of the relationship (referred to as the form of the relationship), a statement of the basis of abstraction, and a statement of how the relationship has been formalized. Relationships are formalized to indicate which instances of one object are associated with the instances of another object. This is done through a referential attribute. For example, given two objects Parent and Child, the referential attribute could be the attribute Parent Name within the Child object. The end result of this stage is the Information Model, and descriptions of objects, attributes and relationships.

2. Constructing the State Model

The lifecycle of the objects is expressed as a state model. The state model is composed of a set of states, which represent stages in the lifecycle of the object, a set of events that represent an incident, transitions that specify the new state achieved after an event has occurred, and actions which are activities or operations to be carried out when an instance arrives at a state. Each state is given a unique name and a unique number within the state model. As this approach uses the concept of the lifecycle of the object, two common states

are the Create and Final states which indicate at which point the instance is created or ends, respectively. Objects generally also have an attribute called current state or status that indicates the state in which an object finds itself at any particular moment.

Events represent an incident or signal in the real world that indicates an object is moving to a new state. They are specified through meaning, destination, label and event data. The meaning is a short phrase that describes what is happening in the real world. The destination is the state model that receives the event. The label, which is unique for each event, distinguishes different events from one another. Event data can take the form of identifier data, which specifies the object instance to receive the event, or supplemental data, which can supply additional attributes of any object that may be required.

Actions are activities or operations that an instance must perform upon arriving in a state. They are provided with the event data carried by the event to perform these operations. Actions can perform calculations, generate events, or read and write attributes of object instances. They must ensure consistency within the object instance and between the objects through the relationships. That is, if an attribute or relationship is changed by an action, the actions must also reflect the changes in dependent objects, attributes and relationships. Descriptions for actions can be placed below the state box or in a separate document.

Transitions are rules that specify what happens, and what new state is achieved, when an event occurs. They are represented by an arrow from one state to another. An example of a transition could be an operator pressing a button to begin a machine. The pressing of the button is the rule that dictates that the machine is being started.

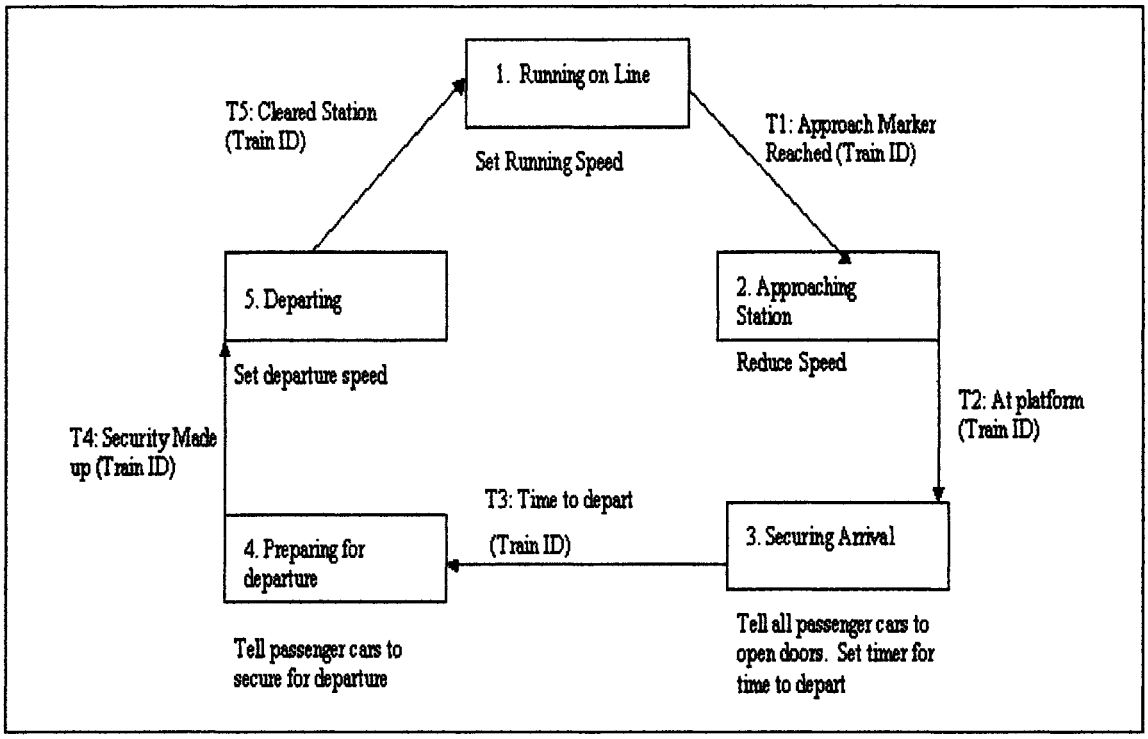

Figure 8: An example of Schlaer & Mellor's state transition diagram (taken from [27])

These four items are meant to fully detail the various events and states. The result is a state transition diagram. An example of a state transition diagram for a train is shown in Figure 8.

3. Constructing the Process Model

This activity focuses on the functional nature of the actions. An action flow diagram (ADFD) is used to represent the units of processing within an action and the communication between them. In the ADFD, processes are represented by circles and are annotated with a process identifier and a meaningful name. Any input data is represented as a data flow to the process, and output data as a data flow from the process. If an event causes an action to be initiated, it is depicted as an event data flow that points into the process from nowhere. Its labels contain the names of the attributes carried by the event and required by the process. Likewise, if a process generates an event, the data flow is directed away from the process, and labeled with the event's meaning, label and event data. Each process is assigned to an object from the information model. This assignment is noted in the process identifier and takes the form of <object>.<arbitrary process>.

The action is partitioned into processes using four defined process types. The first type, *Accessors*, includes processes whose purposes are to access data in a single object store. The accessor may create new instances of an object, read or write attributes of a single object, or delete an instance of an object. *Event generators*, the second type, are processes that produce exactly one event as output. The third type, *transformations*, is a process with the purpose of computing or transforming data. The process converts its input data to a new form that is the output data. A *test* process type tests a condition to activate one of several outputs when a process requires conditional control.

Once the processes have been identified, they are named and described according to the process type. An accessor is named to state what it does and only requires a description if there is some complex selection criteria. Event generators are named "Generate <event label>" and do not require process descriptions. A transformation should be named by the data it produces and the process description can be expressed mathematically, in a narrative, or in a combination of both. Tests are named using "test" or "determine" followed by the name of the condition being tested. A description should be included if the expression being tested is defined or computed within the process.

A listing of the processes is made in the form of a state process table. The state process table is simply a table that lists the process identifier, the process type, process name and in which action it is used. This step ends once the processes have been identified and described.

3.2.6 OBJECT-ORIENTED ANALYSIS AND DESIGN BY MARTIN

Although other OOA techniques are divided into static and dynamic methods, this technique [28] attempts to integrate the two.

1. Define Analysis focus

The first step is to define the realm of interest of the problem. The goal event type, which describes the purpose of the realm, is then identified. The goal event type is the change in object state that should be reached in the realm. The goal event type is the starting point for the next activities.

2. Clarify Event Type

Events are noteworthy changes in the state of an object. Event types are categories of events. Event types are to events as object types (or classes) are to objects. Examples of event types are the creation of an object, the termination of an object, the change of an

attribute of an object or the classification/declassification of an object as an instance of an object type. The event type being examined is clarified by identifying the kind of event to which it belongs. The object types involved in the event are also noted. The event type is then named.

3. Generalize Event Type

The event type, along with the pre-event and post-event states of the object, is generalized (the level of generalization is also decided at this stage) to define whether the event type describes an accurate level of abstraction. The pre- and post-state object types that have been identified are also modified and generalized to the appropriate level of object types. If it is discovered that the generalized event type has already been specified, this event type may be integrated into already specified types.

4. Define Operation Conditions

Once the event type has been specified, the operation that leads to the specified event type can be identified. This operation is examined to determine whether the operation is processed within the realm (internal) or outside of the realm (external). The next step is to identify the control conditions of the operation. Control conditions are conditions that must be checked prior to invoking the operation. An operation may have a single condition or a collection of Boolean conditions. Each of the terms of the control condition will reveal additional objects to be added to the model.

5. Identify Operation Causes

The first step is to determine the event type that triggers the operation. This is called the triggering event type and is the event type that must occur to satisfy the control conditions mentioned above. If different triggering events apply only to certain parts of the control

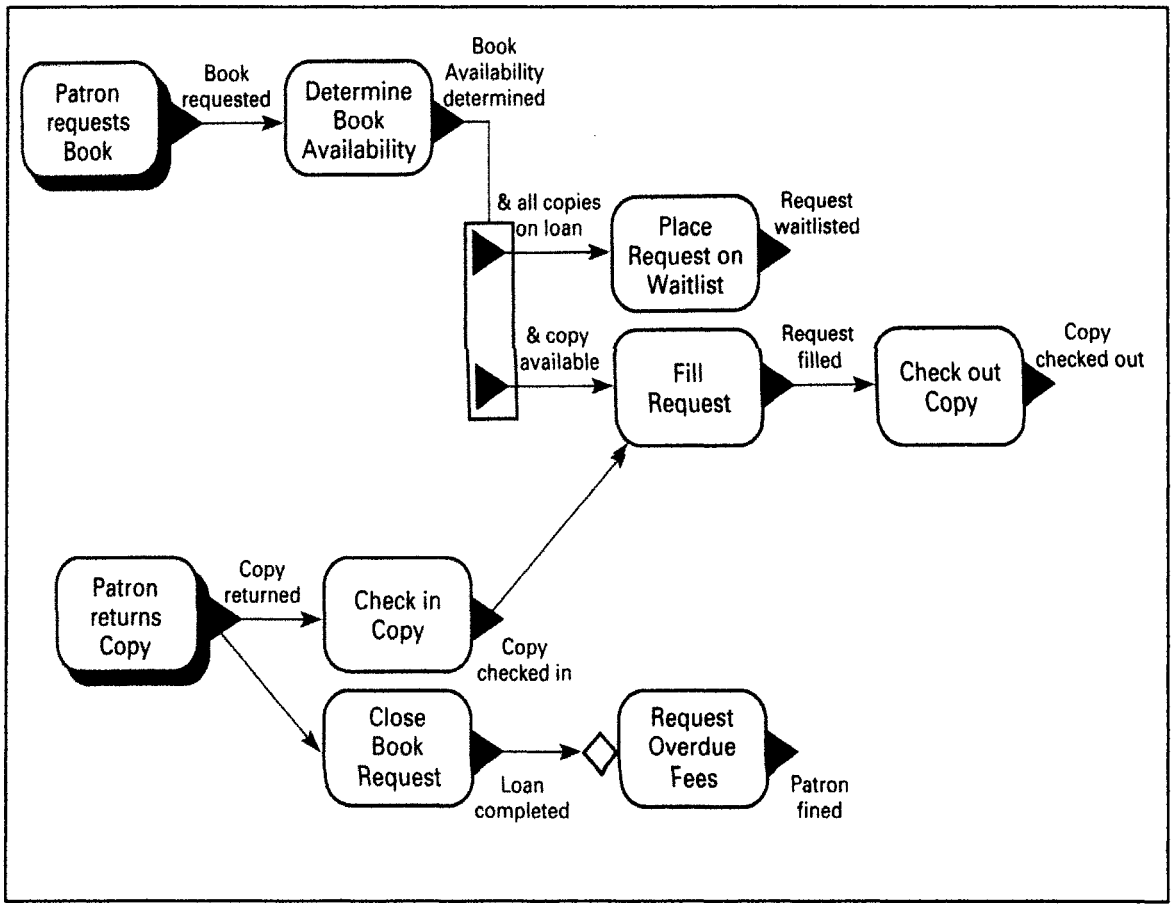

Figure 9: An example of Martin & Odell's event diagram (taken from [28])

condition, the control conditions can be regrouped in several control condition blocks. The trigger rules are then specified. A trigger rule is the causal relationship between the event and the operation. Whenever an event occurs, the trigger rule invokes the pre-defined operation. The identification of trigger rules will specify the objects needed to invoke an operation.

6. Refine Cycle Results

This step seeks to refine the results of the above tasks. Triggering events are checked for generalization into a higher-level event type. Goal events are checked for specialization to clarify their meanings. Triggering event types are compared to goal event types for sameness or sub classification. Duplicates are removed. Steps 2-6 are repeated for each internal event until all such events are specified. An example of a completed event diagram can be seen in Figure 9.

These are just some of the more popular object-oriented techniques. A later section attempts to derive a general object-oriented analysis technique based on similarities in these approaches.

3.3 Unified Modeling Language

The Unified Modeling Language (UML) is a notation for specifying, visualizing and documenting the components of not just software systems but other types of systems as well. The method was developed by Grady Booch, Jim Rumbaugh, and Ivar Jacobson from Rational Software Corporation [29] who combined their individual object-oriented analysis techniques, and included contributions from leading software vendors and users.

The primary goals in the development of UML were as follows [30]:

1) Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.

2) Provide extensibility and specialization mechanisms to extend the core concepts.

3) Be independent of particular programming languages and development processes.

4) Provide a formal basis for understanding the modeling language.

5) Encourage the growth of the OO tools market.

6) Support higher-level development concepts such as collaborations, frameworks, patterns and components.

7) Integrate best practices.

Prior to UML, users chose from many different modeling languages with minor differences. UML has been referred to as the first major international standard in object-oriented modeling [29]. The following figures depict and describe the basic elements of UML that will apply to the object-oriented analysis technique that is described later.

3.3.1 THE CLASS DIAGRAM

The Class Diagram shows the classes and their relationships. The graphical notation for classes is shown in Figure 10. The name of the class is placed in the top rectangle, while the various attributes and operations of the class are placed in subsequent rectangles. The four smaller diagrams on the right illustrate variations for cases containing only attributes or only operations. This notation is slightly modified in Figure 11 for the representation of object instances. The object name, as well as the class to which it belongs, is indicated in the top line. In the attribute location, the values of the attributes are included. The relationship

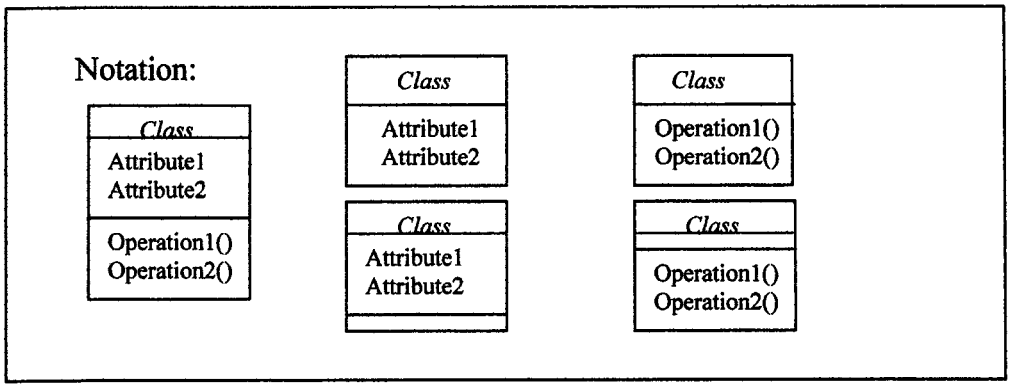

Figure 10: Class Notation in UML

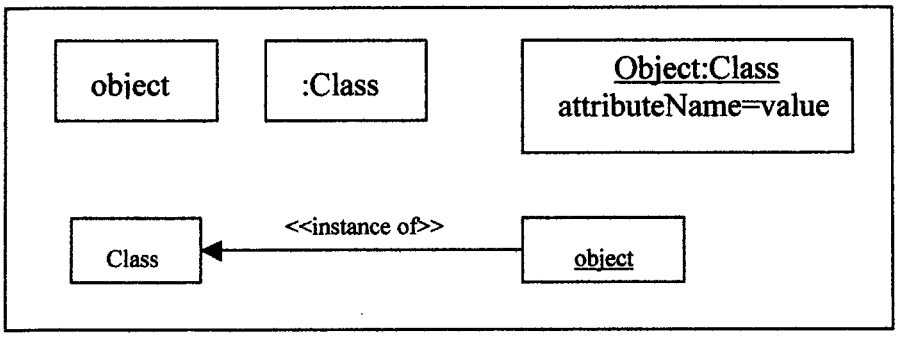

Figure 11: Object Notation in UML

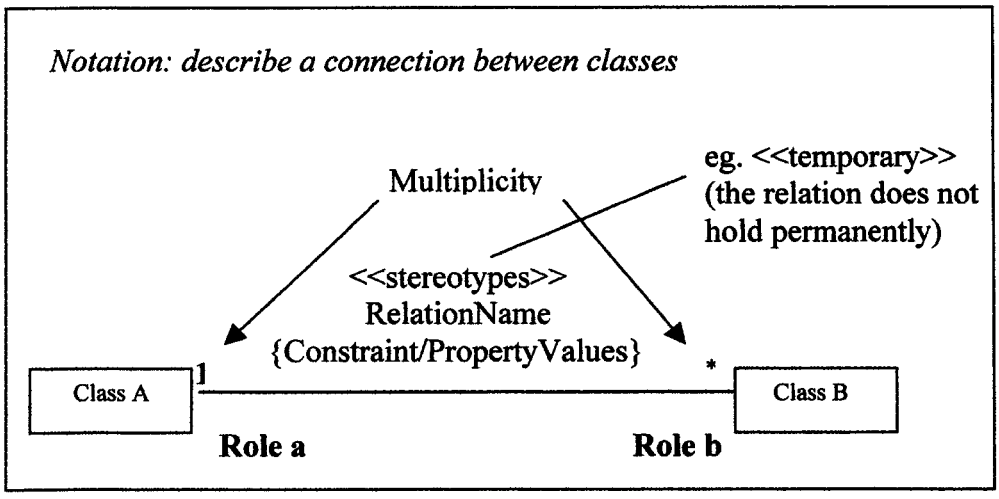

Figure 12: Relationship Notation in UML

between a class and an object can be represented using a labeled directional arrow to show that the object is an instance of the class.

3.3.2 RELATIONSHIPS

UML contains notation for the various relationships between objects. Figure 12 depicts the UML notation for a typical relationship. The two (or more) classes involved in the relationship are represented using the class notation and are joined with a line. The role each class plays is indicated under the line, next to the class. Multiplicity can be indicated above the line, next to the class. In this case, Class A has a one-to-many relationship with Class B. The name of the relationship is placed near the line, in a location that clearly indicates to which relationship it is referring. Any constraints or necessary values can be placed here as well.

3.3.3 USE CASE MODELING

The Use Case Descriptions and Diagrams show the system's use cases and the actors that correspond to each case. The use cases described in the diagram should represent regular occurrences in the system. Any exceptions, or possible error scenarios should not be included in the diagram but may be included in the description. Figure 13 is a UML Use Case Description, which describes the name of the use case, the actors, any necessary conditions, the process, and any error states, can contain as much of this information as is required or relevant. Figure 14 is a Use Case Diagram and is a graphical interpretation of the descriptions. The use case diagram would include several or all of the various use cases developed. The border represents the system boundary. The actors, generally users, are represented either as stick people or in boxes and remain outside the boundary. The various

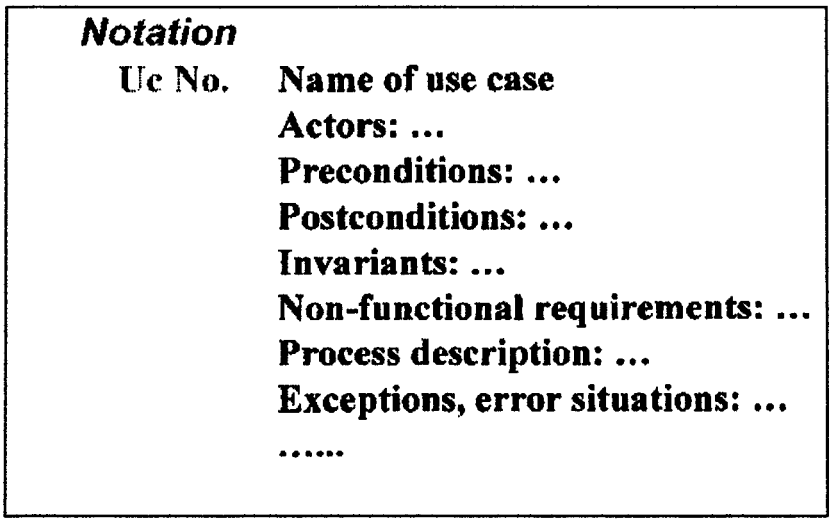

Figure 13: Use Case Description in UML

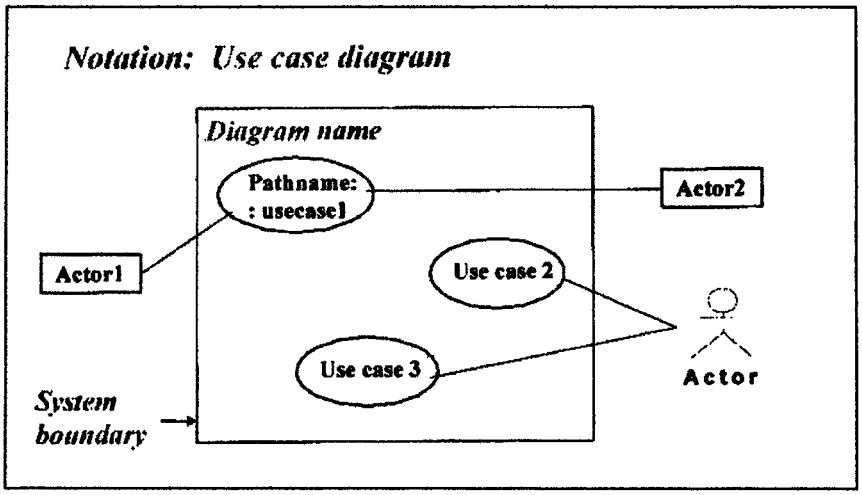

Figure 14: Use Case Diagram in UML

scenarios, or use cases, are represented as ovals within the system boundary. The actors are linked, through the system boundary, to the use cases in which they participate.

The above diagrams cover the very basics of UML. There are many other representations and notations available to describe the many stages of developing software systems. However, only these elements are required for this application.

3.4 General Outline of an Object Oriented Analysis Technique

After examining several object-oriented techniques, this section outlines several steps in order to state in more details the underlying pattern of an OOA technique. Brinkkemper et al [21], in their attempts to compare various OOA and OOD methods, outlined a 'supermethod'. This supermethod was developed by taking the smallest common denominator of activities from various methods, and adding activities deemed to be important. While not an actual method itself, it serves as the basis for a general outline of OOA and OOD approaches. The 'supermethod' is structured in the following way:

1. Analysis : This step includes analysis of requirements, classes and objects, relationships, attributes, subsystems, behaviour, functional architecture/operations, and communication.

2. Design: The second stage investigates the design of specifications, classes and class structures, operations, relationships/attributes, human interactions, data management and the system.

3. Implementation

4. Testing

The supermethod will serve as the basis for this outline. The outline modifies the analysis portion of the supermethod for the purpose of narrowing the scope somewhat. Narrowing the scope is necessary to move from a software development point of view to a method more

applicable to a simulation point of view. The modification results in the following general steps in performing an OOA technique, still in terms of software development.

1. Identify the requirements of the system.
2. Identify classes and objects.
3. Identify attributes and relationships between objects.
4. Identify behaviour (actions) and states of objects.
5. Develop identifiers for objects.
6. Formalize object descriptions.
7. Develop hierarchies and hierarchy graphs.
8. Test/Walk-through.
9. Implementation (which includes object-oriented design, architecture development, coding, etc.).

Because the purpose here is to focus on object-oriented analysis, the Implementation step contains all elements of the techniques that occur once identification of classes and objects is complete. This includes any tasks or issues concerned with designing the software in terms of object-oriented principles, implementing the software, and maintaining it. The next step is to adapt this general layout of an object-oriented analysis technique for simulation. However, since this adaptation will apply to a specific piece of object-oriented simulation software, it is appropriate to take some time to discuss several pieces of commercial object-oriented simulation software here.

Chapter 4: Commercial Object-Oriented Simulation Software

As the field of object-oriented simulation grows, there has been a corresponding development of commercial software based on object-oriented principles. Because this is a relatively new field, there are only a handful of commercial object-oriented simulation tools available. Although this list is not comprehensive, it will give a brief description of several of the commercial software titles available.

4.1 SIMULA

SIMULA [31] was originally built and designed at the Norwegian Computing Centre between 1962 and 1967. Although it is now usually implemented as a full-scale general purpose programming language, it was originally designed as a language for discrete event simulations. SIMULA was responsible for introducing concepts such as object-oriented programming, including the idea of classes and objects. It is considered the first object-oriented simulation language. SIMULA has never become widely used, but has been a tremendously influential modern programming methodology [31]. Most general object-oriented programming languages are subsets of the SIMULA language whose standard was originally defined in 1967. The current SIMULA standard was developed in 1987 [32]. Although Simula is now more of a general programming language than a commercial simulation language, it is included here because of its importance in the development of today's object-oriented simulation languages, and object-oriented programming languages in general.

4.2 Simple++

Simple++ (Simulations in Production, Logistics and Engineering design and the implementation in C++), introduced in 1987, is marketed and supported by Aesop [33]. Simple++ is a standard piece of software for object-oriented, graphical and integrated modeling simulation and animation. This software can be used for a variety of applications such as assembly, warehouses, transportation systems, forecasting, and control. A graphical user interface is available for interactive operation. Unlike other software, separate programs for modeling, simulation and animation are not required. An object template allows all objects to be visible and accessible. There are several key objects in Simple++. The Physical Objects are processors, conveyors, lines, stocks, vehicles, parts, etc. Information Objects indicate the flow of information between objects. The Manager controls the simulation, whereas the Profiler analyzes the run-time behaviour. The Statistics object makes available a variety of statistical functions. A text editor, debugger, graphics editor, and animation are also available.

4.3 MODSIM III

MODSIM III, by Caci Products Company [34], is a general-purpose simulation tool, developed in the early 90s, for modeling large, complex systems. It fully supports object-oriented programming, including multiple inheritance, encapsulation, and polymorphism. MODSIM III has been applied to defense modeling, network simulation, transportation modeling, supply chain simulation, and business process modeling, among others. In MODSIM III, objects are described in two separate blocks of code: the Definition block and the Implementation block. In the definition block, the object is described through the declaration of its variables and methods. This is the object description that other objects in

the simulation will refer to, and so forms the interface. The methods are described in the Definition block, but the logic of what they do and how the variables are affected, is described in the Implementation block. MODSIM III offers a collection of simulation building block objects, as well as graphics libraries for graphical scenario layout. MODSIM III forms the basis of Caci's process simulation tool, SIMPROCESS.

4.4 Silk

Silk was developed in the late 90s by ThreadTec, Inc. [35] and was chosen as the object-oriented simulation language for this research. It was selected because it combines the power of Java-based programming with familiar process-oriented modeling tools, offering the best of both the traditional and object-oriented simulation worlds. Silk is based on the Java programming language and so this section will begin with a short discussion on Java based simulation languages. A more in depth description of Silk will then follow.

4.4.1 Java Based Simulation

A Java based simulation is simply a simulation programmed in the Java language [35]. Silk is a general-purpose simulation language based on the Java programming language. Therefore, it is appropriate to discuss the benefits of using Java for simulation. Java offers several benefits for modelers and simulation. Java is the only object-oriented programming environment that effectively supports standardized components [36]. This standardization is most significantly realized through Java's ability to handle multi-threaded programming. Multi-threaded programming enables the creation of an entity-thread that has independent control of its behaviour within the simulation. In most programs, there is a single thread, that is, there is a single intelligent entity that is controlling the execution of each instruction. Multiple, independent, intelligent entities share control of the execution of instructions in

multi-threaded environments. This type of environment better simulates the real world, where there are several intelligent entities making decisions. In reality, if there is only one processor running the program, only one thread is active at any time and the other threads are suspended until specified conditions are met. In essence, instead of using a programming language to construct a simulation language that emulates a set of tasks, Java (the programming language) **is** the simulation language. Thus, using Java, all simulation tools could share a common set of standards and developing simulation programs would simply consist of assembling reusable and compatible simulation components based on the standards.

The existence of third party Java development environments, and trained users for those developments, allows more time and resources to be devoted to the development of application specific simulation components. Simulation vendors would no longer need to worry about spending valuable resources on integrated development environments, graphical user interfaces, and animation and other input/output options.

Java can also expose the benefits of computer simulation to a larger audience because the models can be distributed and executed using standard browser software over the Internet. Simulation education can also be enhanced since students already familiar with object-oriented design and Java will not have to learn specific simulation tools [7].

4.4.2 Description of Silk

As mentioned above, Silk is a Java-based general-purpose simulation language. Silk is more of an extension of the Java language than a language in itself [37]. It represents a marriage of process-oriented modeling and the object-oriented features of a programming language [37]. Silk may be seen as a number of things. It is a collection of Java class

libraries that can be assembled into a variety of modeling constructs. It is also a process-oriented simulation language that still provides the opportunity to program in a standard programming language.

Silk allows the development of simulation models directly in Java using a package of classes that provide process-oriented, modeling features. Silk provides the simulation entities, resources and queues as classes. The modeler then develops problem-specific objects by extending these classes. Animation is provided through JavaBeans which is a neutral component architecture for Java applications that provides a user interface. Other classes are available for probability distributions, statistics collection and output generation.

A Silk simulation is composed of many classes. Two classes are required, and other classes are added as necessitated by the given situation. One required class is a distinguished class and defines the point of entry at which the program begins executing. This class must create an instance of the predefined class *Silk* and invoke its *begin()* command. This essentially begins the simulation. The other required class, called *Simulation,* has as one of its methods *run()* which serves as the starting point of execution for the model. Initial and final conditions, such as creating entities and replication information, as well as global model specific data can be described in this class.

A typical third class for a discrete manufacturing system may be called *Part,* which represents the product being produced by the system. Each *Part* object is an extension of the Silk *Entity* class. These object instances run as separate threads of execution (hence the multi-threaded execution) that are alternately suspended and resumed to coordinate the time-ordered sequence of entity movements. A thread is suspended when the corresponding entity encounters a wait at a *delay()* or a status delay at the *while(condition())* construct. The

management of simulated time and the resumption of suspended threads are coordinated by an executive thread running in the background.

The class definition begins with declarations of the data that represent the state of the entity types (i.e. the parts). Variables such as system arrival time are defined here, as well as the various resources, queues, or other objects the entity may encounter during the process. Other possible objects for this section include Silk classes used to record time-weighted statistics on queue length and utilization, and observational statistics on the time entities spend in the system.

The second part of class definition involves identifying class behaviours that are implemented as methods. Methods are analogous to executable functions in traditional procedural programming languages. The formalism in Silk requires that each entity class definition contain at least one distinguished method named *process()* which serves as the starting point of execution for each instance that is created. In this section, the process through which the object moves is described through, among others, a series of queue, seize, dequeue, delay and release commands. These three (or more) class files are then compiled together to execute the simulation. Silk requires extensive programming, and while this may make it more difficult for new users, it allows for a more powerful simulation. Silk will be used to test the object-oriented analysis procedure for systems to be simulated that is described in a later section.

Chapter 5: Proposed Object Oriented Analysis Technique for Simulation

The object oriented analysis techniques and approaches discussed thus far have been based upon, and applied to, software development and engineering. That is, they were intended to analyze a given system and identify the objects within that system so that an appropriate piece of software could be developed using the object-oriented programming paradigm. Their concepts and activities are meant to form the first steps in the full-fledged development of software. As this is the only situation in which they've been applied, and for which they have been developed, the techniques and approaches possess a strong "software development" flavour. However, it is the assertion of this thesis that these same techniques and approaches can be applied to simulation. By applying these techniques to a system to be simulated, a simulation analyst new to object-oriented simulation will be better able to model their system in an object-oriented simulation language and take advantage of its benefits.

This section will describe how the general layout of object-oriented analysis techniques, described in Section 3.4, has been adapted and modified for the purpose of understanding a manufacturing system in terms of its objects with the goal of developing a discrete event simulation model for that system. This understanding is integral to developing object-oriented simulations since objects form the basis of an object-oriented program. As will be seen, translating the techniques from pure object-oriented analysis techniques to those applied to simulation requires changes in some activities, to some concepts, and to some underlying reasons behind the concepts. This is mainly caused by the difference in the goals of the application of object-oriented analysis. That is, current object-oriented analysis techniques seek to assist in building and developing software. In the process of applying

them to simulation, one notes that the software already exists. However, the principles of object-oriented analysis can be used to identify the appropriate objects in the given manufacturing system, and better understand how these objects interrelate.

The motivation here is to provide a method for those new to the object-oriented programming paradigm to apply to an object-oriented simulation tool. It, therefore, focuses on the noun extraction technique, which has been identified as a good approach for those without expertise [9]. To summarize, the noun extraction technique involves describing a system in full sentences. The various parts of the sentences (e.g. nouns, verbs, etc.) are then mapped to various objects and object components (classes, relationships, etc.). In the context of the simulation process, this set of guidelines would lead off the "Building the Model" phase.

An important note regarding jargon should also be made here. As was explained in Section 2.2, strictly speaking, classes are a group of objects and provide the pattern for objects. Objects are the run-time entities themselves. For example, in the simulation of an automotive assembly plant, *Car* would be the class, and the individual cars moving through the system *Car 1, Car 2, Car 3, etc.*, would be the objects. However, this technique will, for the sake of simplicity, extend the meaning of the word object and use it to represent some classes as well. Thus, a single machine will be called an object (as it normally would), but the part being produced will also be called an object (even though it is, strictly speaking, a class). Any item within the scope of the system will be referred to as an object. Ignoring the nuances of the two definitions allows for greater flexibility and applicability of the technique. The technique does not need to become bogged down by concerning itself with the number of instances each object/class being created at this early stage. It also allows the technique to

be applied to any type of manufacturing system, whether the parts or the resources are active or passive.

The steps for object-oriented analysis for simulation are :

1. Identify the scope of the system.
2. Describe the system in sentences.
3. Use nouns to identify objects.
4. Use nouns with possessive phrases to identify attributes.
5. Use verbs to identify relationships.
6. Perform use-case modeling to identify remaining objects.
7. Identify desired measurables (e.g. part count, jobs per hour) as objects.
8. Identify "Entity"objects (specific to Silk).
9. Review and refine objects.
10. Match remaining objects to Silk classes.

The steps will now be explained in more detail.

Step 1: Identify the scope of the system

This first step is common in any problem solving activity and may have been performed in an earlier step of the simulation process. If it hasn't been performed, the scope can be identified at this point. If it has, this opportunity can be taken to verify the scope after the collection of data (phase two of the simulation process) since it will be used in the second step. In any simulation it is very important to define what is being included in the study. The scope may be a statement, a diagram, or some other easily understood form.

Step 2: Describe the system in sentences

As mentioned, this technique uses the concept of noun extraction. The first stage of noun extraction is to describe the system in sentences. Thus the system being simulated should be described in full sentences and should include detail for components within the scope of the system. According to the noun extraction method, these sentences should be as concise as possible, while still describing the entirety of the system and its constraints. While flying in the face of a picture being worth a thousand words, and the intuition that a good diagram is all the explanation needed, this written description forms the basis for the noun extraction method. Therefore, the tedious nature of describing the system in sentences will benefit the novice user later in the actual identification of objects.

Step 3: Use nouns to identify objects

Noun extraction is a recognized method of identifying objects, attributes and relationships [12]. In this step, the written description of Step 2 is analyzed for nouns. These nouns will form a candidate list of objects for later use. As the purpose here is to identify objects, the repetition of nouns is unnecessary. If *Machine X* is referred to three times, it need only be noted on the list of candidate objects once. The identification of objects is directly related to the implementation of the model, since the model is built through the programming of objects.

Step 4: Use nouns with possessive phrases to identify attributes

Any nouns with possessive phrases may represent attributes of the objects. This candidate list is also reserved. Although the identification of attributes is directly related to the implementation of the model, it also assists in better understanding the relationships

between system components. Depending on the attribute, it may represent a calculation to be performed using another object, or information that must be obtained.

Step 5: Use verbs to identify relationships

In the noun extraction method, verbs represent relationships between objects. Although the proposed technique takes this method of identifying relationships from software development, the concept of relationships is slightly different in the case of simulation as compared to software development. In software development, the relationships are used to place the object in the class hierarchy and study how the objects relate to each other so as to combine them in such a way that the software will be functional and efficient. In the case of simulation, the relationships will simply help the model builder better understand how the objects of the system interact, thereby providing insight as to how they should be mapped to Silk objects and how the simulation model should be constructed.

Step 6: Perform use-case modeling to identify remaining objects

This step employs use-case modeling. This concept is adapted here, with the "user" being the part being produced and the scenarios (and functionality) being the possible scenarios the part can encounter while moving through the system. These scenarios will help to uncover various statistics and variables that should be measured to provide an accurate picture of system behaviour. As opposed to software development, when applying use case modeling to simulation, somewhat frequent exceptions and error states should be included to better understand how they affect the system. In software development, these error states and exceptions are excluded because the purpose is to establish the intended functionality. However, in simulation, they are required to gain a true insight into how the system operates and to identify any statistics or variables that may need to be included and collected. Any

desired statistics or variables should be added to the candidate object list (in the case of statistics) or a separate variable list.

Step 7: Identify desired measurables as objects

The noun extraction method of object-oriented analysis may not cover all needed calculations or identify all information to be collected. Step 7 is a review phase that is based on the experience of the analyst. The analyst should take this opportunity to ensure that all measurables for the system up to this point have been included, and add any that are necessary. This may include part count, jobs per hour, or any other statistics deemed important. This step acts as a reminder of other items the modeler may want to measure. These objects should also be added to the candidate object list.

Step 8: Identify "Entity"objects (specific to Silk)

This step works with Step 5 to identify the remaining key class(es) of Silk, that is, the Entity class. As was previously discussed, any Silk simulation contains a *Simulation* class and a distinguished class, which usually carries the name of the Simulation. Usually, the third class in a Silk simulation is the Entity class. Entity classes generally represent the items flowing through the system. Although in many cases it may be obvious, this step will aid in determining that the flow of the proper item is being modeled. Step 5 assists the analyst in determining which one type of object has the most interaction with other types of objects. It is important to note that there may be more than one Entity object in a system, as will be seen in the example in the next chapter. It also must be stated that Silk simulations can contain classes besides the three types (Simulation, distinguished class, and Entity) described here, if necessary. These three types of classes simply cover the most general case.

Step 9: Review and refine objects

Almost all object-oriented analysis techniques follow some sort of iterative pattern. As the analyst becomes more familiar with the system at hand, it becomes clearer as to which object classes are relevant and appropriate. This stage is a review process to determine which, if any, objects need to be eliminated or added. The candidate list that has been developed to this point is analyzed to determine the objects to be included, and superfluous or unnecessary objects should be eliminated. Candidate attributes and relationships can also be re-examined at this stage to ensure that system interactions have been represented accurately.

Step 10: Match remaining objects to Silk classes

This step uses the model builder's experience to map remaining objects (i.e. those other than the "Entity" class object mapped in Step 9) to the available objects in Silk (e.g. resource, queue, etc.) Once the system has been decomposed into its objects, and fully understood by the analyst, this activity becomes a reasonably straightforward mapping.

This approach allows the modeler new to object-oriented techniques to study the given system in terms of objects and match those objects to specific classes in Silk to build the simulation. Given the lack of typical drop-down-box, fill-in-the-space user interface, it also allows the slightly more experienced user to ensure that all system components to be modeled, measured, or timed are included. The next section goes through an example of this approach.

Chapter 6: Example

6.1 The System

The system used for this example is taken from Alami [38]. Some minor aspects of the process plan have been modified. The system is a manufacturing cell environment consisting of 3 cells, 14 machine types, and 10 part families. A schematic can be found in Figure 15. This system uses the concept of alternative machines, which are to be used when the original machine is busy. Cell 1 contains five machines, numbered 1 through 5. Cell 2 contains six machines, numbered 6 through 11. Cell 3 consists of machines 12, 13, and 14, which are usually, though not in all cases, used as alternative machines. According to the example, Cell 3 is meant to serve as a remainder cell. The part families, batch sizes, routings and processing times can be found in Table 1. Intercell transport times, shown in Table 2, are included as well.

The percentage of incoming orders each part family comprises can be seen in Table 3. Because the purpose here is to test the method described in Chapter 5, scheduling issues were not considered. It was assumed that an infinite supply of orders was available. The number of each type of part family entering the system was determined using statistical distributions matching the percentages in Table 3. In [38], it was determined that an optimal order size was approximately twenty shifts, so each simulation run lasted twenty shifts, 480 minutes each shift (9600 minutes), and fifty replications were performed.

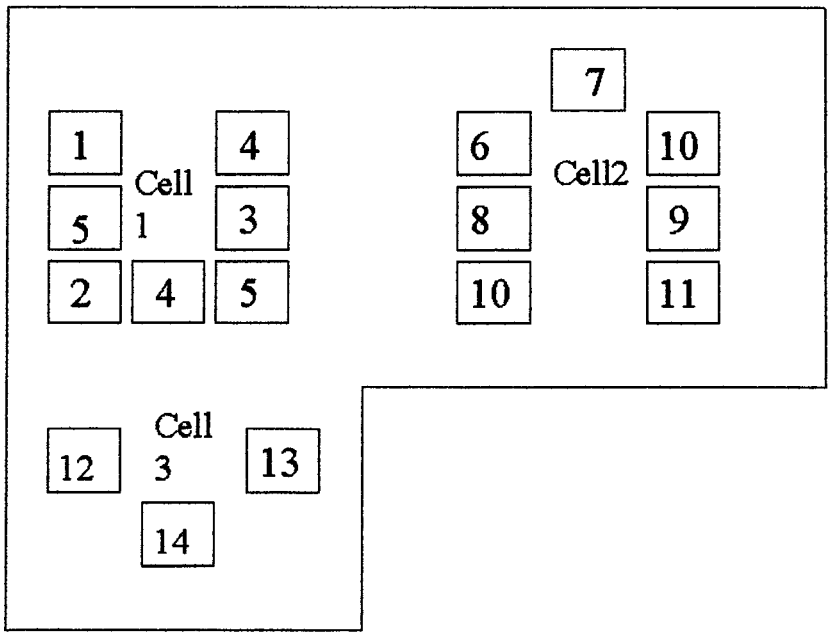

Figure 15: Schematic Layout of Example System

Table 1: Part Families' Routings and Processing Times

Part Family	Batch	Machine		ProcessingTime/part (min)	
	Size	Original	Alternate	Original	Alternative
1	10	1	13	1.6	3.6
		3	----	0.8	----
		5	12	2.1	4.1
		4	6	1.4	1.4
2	75	2	3	0.8	3.8
		1	13	0.6	2.6
		5	12	0.6	2.6
		4	6	1	4
3	24	6	----	1.4	----
		10	----	1	----
		9	----	0.4	----
		8	11	0.8	0.6
4	36	3	----	1.6	----
		1	13	0.8	2.8
		5	----	2.4	----
		4	6	1.4	1.4
		14	----	0.4	----
		12	----	3	----
5	23	6	9	1.2	2.2
		8	11	1.8	0.8
		10	----	0.8	----
		7	14	1	2
6	12	2	3	1.2	4.2
		4	6	1.4	1.4
		5	----	0.8	----
		13	----	2	----
		12	----	1.8	----
7	20	14	----	1.6	----
		9	----	2.6	----
		8	11	1	0.8
		7	----	1.4	----
8	15	1	13	2.8	4.8
		5	12	0.8	2.8
		3	----	1.2	----
		2	----	2.4	----
		11	----	1.6	----
9	19	7	----	3.2	----
		8	11	1	0.8
		10	----	0.4	----
		14	----	2.8	----
10	10	13	----	11.2	----
		6	9	2.9	2.9

Table 2: Intercell Transfer Times (min)

From \ To	Cell 1	Cell 2	Cell 3
Cell 1	----	10	10
Cell 2	10	----	18
Cell 3	10	18	----

Table 3: Part Family Composition of Incoming Orders

Part Family	Batch Size	Percentageof Incoming Orders
1	10	8%
2	75	27%
3	24	8%
4	36	6%
5	23	28%
6	12	5%
7	20	4%
8	15	5%
9	19	6%
10	10	3%

6.2 Application of Proposed Object-Oriented Analysis Technique

Step 1: Identify system

The scope of the system was described in Section 6.1 above.

Step 2: Describe the system in sentences

This flexible manufacturing system contains 3 manufacturing cells, 10 part families, and 14 machines. Part 1 seizes and is processed on Machine 1 with Machine 13 as the alternate, then Machine 3, then Machine 5 with Machine 12 as the alternate, and finally Machine 4 with Machine 6 as the alternate. Part 2 seizes and is processed on Machine 2 with Machine 3 as the alternate, then Machine 1 with Machine 13 as the alternate, then Machine 5 with Machine 12 as the alternate, and finally Machine 4 with Machine 6 as the alternate. Part 3 seizes and is processed on Machine 6 with Machine 9 as the alternate, then Machine 10, then Machine 9, and finally Machine 8 with Machine 11 as the alternate. Part 4 seizes and is processed on Machine 3, then Machine 1 with Machine 13 as the alternate, then Machine 5, then Machine 4 with Machine 6 as the alternate, then Machine 14, and finally Machine 12. Part 5 seizes and is processed on Machine 6 with Machine 9 as the alternate, then Machine 8 with Machines 10 and 11 as alternates, then Machine 10, and finally Machine 7 with Machine 14 as the alternate. Part 6 seizes and is processed on Machine 2 with Machine 3 as the alternate, then Machine 4 with Machine 6 as the alternate, then Machine 5 as the alternate, then Machine 13, and finally Machine 12. Part 7 seizes and is processed on Machine 14, then Machine 9, then Machine 8 with Machine 11 as the alternate, and finally Machine 7 with Machine 14 as the alternate. Part 8 seizes and is processed on Machine 1 with Machine 13 as the alternate, then Machine 5 with Machine 12 as the

alternate, then Machine 3, then Machine 2 with Machine 3 as the alternate and finally Machine 11. Part 9 seizes and is processed on Machine 7 with Machine 14 as the alternate, then Machine 8 with Machine 11 as the alternate, then Machine 10, and finally Machine 14. Part 10 seizes and is processed on Machine 13, and then Machine 6 with Machine 9 as the alternate.

Step 3: Use nouns to identify objects

In this example the candidate list of objects identified through noun extraction would include: *Parts 1-10, Machines 1-14, manufacturing cells,* and *flexible manufacturing system.*

Step 4: Use nouns with possessive phrases to identify attributes

In this example, there are no attributes.

Step 5: Use verbs to identify relationships

The relevant verb tenses in the above description are *seizes* and *is processed on.* Figure 16 depicts the generic relationship using UML notation. Part family *i* (where *i* spans from 1-10) seizes Machine *j* (which spans from 1-14), while Machine *j* processes Part family *i.*

Step 6: Perform use-case modeling to identify remaining objects

A study of possible scenarios for this system yields the two possible use cases shown in Figure 17 in UML notation. A part may follow its original process, or it may use one or more alternative machines.

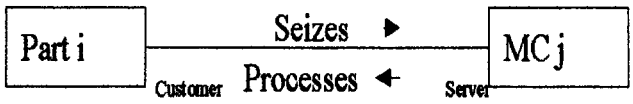

Figure 16: Object Relationships

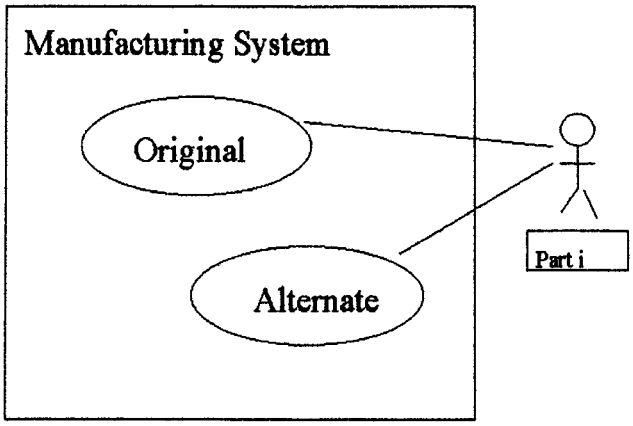

Figure 17: Use Cases for Example

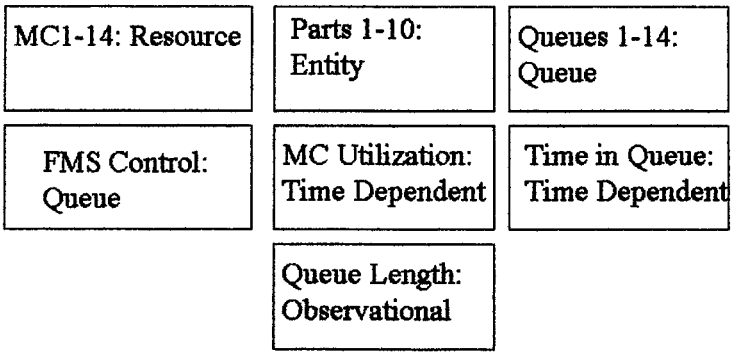

Figure 18: Objects matched to Silk classes

From these use cases, queue objects for each machine will be added to the candidate object list. These queues will assist in determining whether the part follows the original or alternate path.

Step 7: Identify desired measurables as objects

Silk automatically calculates part count during the simulation run. However, given the many queues and machines in the system, statistical objects will be required to measure queue length and machine utilization. Time in system will also be measured. These additional objects should be added to the candidate object list.

Step 8: Identify "Entity"objects (specific to Silk)

Through examination of the candidate objects, and the relationships in Step 5, it can be seen that the appropriate "Entity" objects in this example would be the ten different part families. Therefore, the simulation will have ten "Entity" objects, each one representing a part family and its process.

Step 9: Review and refine objects

The candidate object list now contains: *Parts 1-10, Machines 1-14, manufacturing cells, system, Queues 1-14, QueueLength 1-14, MachineUtilization 1-14, and TimeinSystem 1-10.* Upon examining these objects, it can be seen that *manufacturing cells* serves only to provide boundaries and a layout for the example. Thus, *manufacturing cells* can be eliminated from the candidate object list. *Flexible manufacturing system* dictates when the entity will enter and leave the area of study, based on the availability of machines relative to the entity's process plan. The basic entering and leaving functions are carried out by Silk's *create* and *dispose* statements; therefore, generally, *system* would be eliminated from the candidate list. However, because it is a flexible manufacturing system, some component

must be included to incorporate a level a control that decides which part will enter based on machine availability. The candidate list now contains the parts, FMS control component, machines, queues, and statistics.

Step 10: Match remaining objects to Silk classes

Step 8 matched the *Parts 1-10* objects to Silk's "Entity" class. This step will match the remaining objects to Silk's classes. Figure 18 shows, in UML notation, the matching of these objects to Silk classes.

6.3 Simulation Results and Statistical Analysis

The system described in Sections 6.1 and 6.2 above, was simulated using the Silk software, following the proposed methods outlined in Chapter 5. It was also simulated in Witness, produced by Lanner [39], which uses a more traditional programming paradigm, for comparison purposes. This section will summarize the results of these simulations. Any item labeled "object" refers to results from the object-oriented Silk simulation. Any item labeled "traditional" or "trad" refers to results from the Witness model. In the part mix graphs, "original" refers to the part mix originally entered into the model. An analysis of variance, and a t-test, statistical tests designed to test for similarity of population variances and means, respectively, were performed on each data category to determine if significant statistical differences exist between the object-oriented and traditional models, using an alpha value of 0.05.

Figure 19 is a graph of the percentages of each part family in the part mix entering the system. It can be seen that, in general, both the object-oriented and traditional models follow the original statistical distribution fairly closely. Any differences among the three can be attributed to internal manipulation and administration of the distribution. That is, each model

would generate different random numbers, and sample the distribution in different ways, creating the small differences. Figure 20 is a graph of the same for parts processed (or finished). As can be seen from this graph, the part mixes for both the object oriented and traditional simulations roughly follow the original distribution as well. Given the lack of real world system with which to validate the models, this study of the part mixes simply serves as a preliminary analysis to determine the models are operating properly. Differences between the object oriented model's processed part mix and the traditional model's processed part mix may be a result of the differences between the two types of models in process logic and model structure. These differences will be described later when differences between other model statistics are discussed.

Table 4 summarizes the throughput results for each model. The averages and confidence intervals are shown, as well as whether or not there was a statistical similarity between the two methods. For this measurable, part families 2, 3 and 8 produced statistically similar numbers using both models. Statistically similar results were obtained from the two models for three out of ten part families, and the remaining part families had differences between the two models ranging from approximately 2-14 parts.

Table 5 summarizes the time each part spent in the system. There are significant statistical differences between the object-oriented and traditional models for all part families, though visual inspection reveals that, while statistically different, the results are often generally similar.

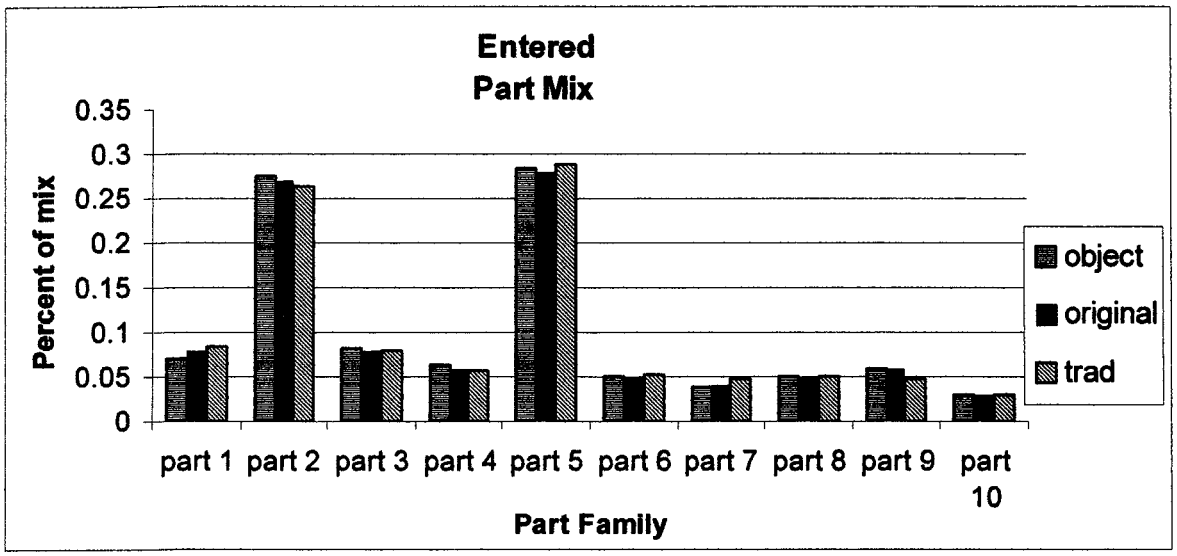

Figure 19: Graph representing the percentage part mix for entering parts for the two models and the original mix.

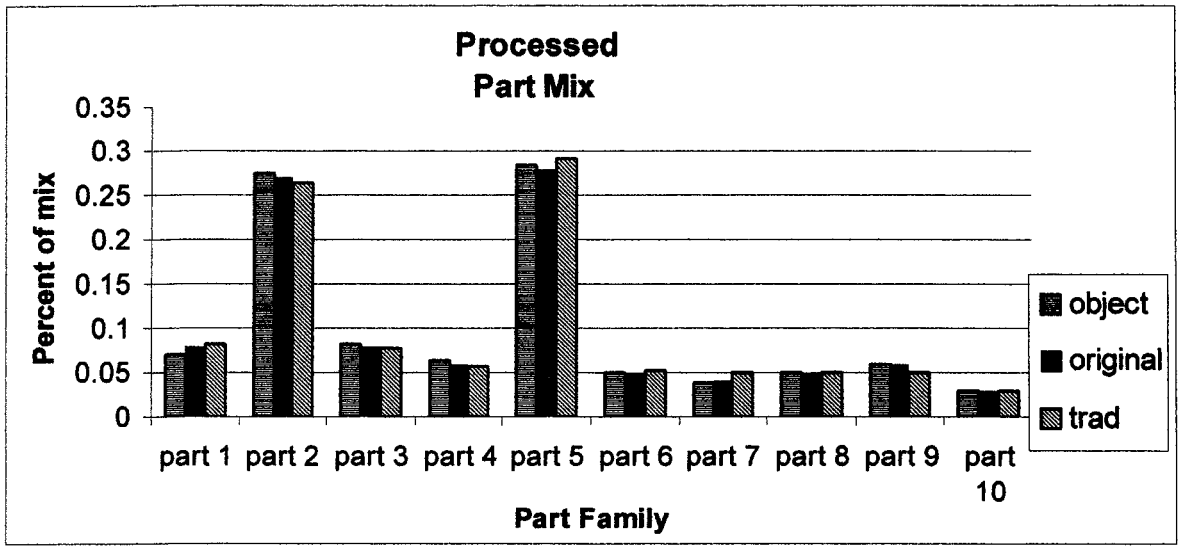

Figure 20: Graph representing the percentage part mix for processed parts for the two models and the original mix

Table 4: Summary of Results for Number of parts of each Part Family Processed

Identifier	Object	Traditional	Stat Same?	95% Confidence Interval for OO	95% Confidence Interval for Traditional
Number of Part Family 1 Processed	44.8600	54.9400	no	(43.264, 46.456)	(54.822, 55.058)
Number of Part Family 2 Processed	174.9800	176.9800	yes	(171.202, 178.758)	(176.746, 177.214)
Number of Part Family 3 Processed	52.0800	52.0000	yes	(50.524, 53.636)	(51.99972, 52.00028)
Number of Part Family 4 Processed	39.8400	38.0000	no	(38.268, 41.412)	(37.99972, 38.00028)
Number of Part Family 5 Processed	181.0200	195.5800	no	(177.936, 184.104)	(195.442, 195.719)
Number of Part Family 6 Processed	31.5200	35.0000	no	(29.752, 33.288)	(34.9972. 35.0028)
Number of Part Family 7 Processed	24.2200	33.0000	no	(22.847, 25.593)	(32.99972, 33.00028)
Number of Part Family 8 Processed	32.3200	33.5800	yes	(30.800, 33.840)	(33.442, 33.719)
Number of Part Family 9 Processed	37.5800	33.0000	no	(36.211, 38.950)	(32.9972, 33.00028)
Number of Part Family 10 Processed	18.1200	20.0000	no	(16.930, 19.310)	(19.9972, 20.00028

Table 5: Summary of Results for the Average Time (min) each Part spent in the System

Identifier	Object	Traditional	Stat Same	95% Confidence Interval for OO	95% Confidence Interval for Traditional
Part Family 1 Time in System	7.9386	16.1908	no	(7.924, 7.953)	(16.181, 16.201)
Part Family 2 Time in System	6.5416	15.5906	no	(4.793, 8.290)	(15.587, 15.595)
Part Family 3 Time in System	3.5528	15.3886	no	(3.548, 3.558)	(15.349, 15.429)
Part Family 4 Time in System	20.4717	17.2621	no	(19.820, 21.123)	(17.255,17.269)
Part Family 5 Time in System	6.0685	15.5104	no	(4.228, 7.909)	(15.508,15.513)
Part Family 6 Time in System	17.9457	17.3300	no	(17.840, 18.051)	(17.3297,17.3303)
Part Family 7 Time in System	16.8017	17.5800	no	(16.695, 16.909)	(17.541,17.619)
Part Family 8 Time in System	19.6399	17.8960	no	(19.448,19.832)	(17.882,17.910)
Part Family 9 Time in System	17.3181	17.7420	no	(17.149, 17.487)	(17.738,17.746)
Part Family 10 Time in System	13.1000	16.9136	no	(13.0997,13.1003)	(16.903,16.924)

Table 6: Summary of Results for Machine Utilization

Identifier	Object	Traditional	Stat Same	95% Confidence Interval for OO	95% Confidence Interval for Traditional
Machine 1 utilization	0.029058	0.030214	no	(0.0285, 0.0296)	(0.03018, 0.03024)
Machine 2 utilization	0.028479	0.027520	yes	(0.02445, 0.03251)	(0.027507, 0.027533)
Machine 3 utilization	0.021760	0.014846	no	(0.01534, 0.02818)	(0.01431, 0.01538)
Machine 4 utilization	0.035193	0.018499	no	(0.03459, 0.03579)	(0.018487, 0.018511)
Machine 5 utilization	0.053740	0.016385	no	(0.03734, 0.07014)	(0.016376, 0.016394)
Machine 6 utilization	0.032724	0.048646	yes	(0.03196, 0.03349)	(0.03965, 0.066327)
Machine 7 utilization	0.034976	0.056204	yes	(0.03436, 0.03559)	(0.03457, 0.07784)
Machine 8 utilization	0.043190	0.064208	yes	(0.04259, 0.04379)	(0.0409, 0.0875)
Machine 9 utilization	0.012219	0.022704	no	(0.01176, 0.01267)	(0.01368, 0.03173)
Machine 10 utilization	0.022077	0.024170	yes	(0.02177, 0.02239)	(0.01469, 0.03365)
Machine 11 utilization	0.007799	0.006116	yes	(0.00539, 0.01021)	(0.00609, 0.00614)
Machine 12 utilization	0.018360	0.073984	no	(0.01775, 0.01897)	(0.04712, 0.10084
Machine 13 utilization	0.019133	0.100066	no	(0.014258, 0.02401)	(0.09647, 0.10366)
Machine 14 utilization	0.019636	0.127800	no	(0.01449, 0.02478)	(0.1278, 0.1278)

Table 6 summarizes the average machine utilizations for the fourteen machines in the two models. Utilization rates for Machines 2, 6, 7, 8, 10 and 11 are statistically the same. The utilizations for machines 1, 3, and 9 are within a couple percentage points of each other, while Machines 13 and 14 show the largest disparity between the two models. The traditional model appears to be using the alternate machines 12, 13 and 14 more often than the object-oriented model. Machine queue lengths were all either zero or of a magnitude so small (e.g. 10^{-4}) that no type of statistical analysis is given.

Discussion of Statistical Analysis

Three key statistics were measured to determine if the two models produced similar results: parts processed, machine utilization, and time in system (for each part family). Three out of ten part families produced statistically similar results, in terms of number of parts processed, from the two models. Six out of fourteen machines had statistically similar utilization rates. Upon further visual inspection of the remaining measurables, it can be seen that, three of the remaining seven "parts processed" statistics (part families 4, 6, and 10) and three of the remaining eight "machine utilization" statistics (machines 1, 3, and 9), while statistically different, have average values from the two models that fall within a rather reasonable, acceptable range of values. It does appear that the traditional model had a higher utilization rate of Machines 12, 13 and 14. It is believed that this higher utilization rate, and the statistical differences of the remaining "parts processed" and "machine utilization" statistics are related to differences in the internal workings of the object-oriented model and the traditional model.

The two models were built using the same process logic. The model logic was verified several times to ensure the models were built using the same rules, and logic. Part

distributions entering and leaving the model were graphed to validate the models. In each model, parts enter the system according to a user-defined distribution that mirrors the part family distribution given. Given that two different types of software are being used, the distributions are defined in slightly different ways, but both produce results that match the original distribution. In accordance with flexible manufacturing system principles, each part then waits for its first machine to be available before entering the manufacturing system. Most of the processes have an alternate machine, so each part, at each stage, checks if its original machine is available first. If the machine is available, the part seizes that machine; if the machine is not available, the part is sent to the buffer or queue of the alternate machine. Again, while the format may be slightly different in each model, the logic is the same in both. The part continues to move through the system in this manner, following its own process plan, until it is finished, and removed from the system.

Two of the measurables, "parts processed" and "machine utilization" had several statistics that were determined to be statistically similar, with many more within an reasonable range to accept that the models are similar. However, all "time in system" statistics were statistically different and in general, the "time in system" statistics of the traditional model were larger than the time in system of the object-oriented model. The increased use of the alternate machines 12, 13, and 14 in the traditional model would, in part, explain the larger "time in system" statistics for the traditional model. Machines 12, 13, and 14 were grouped in a third cell. Intercellular travel times between the cells (from Cell 1 to Cell 3, and from Cell 2 to Cell 3) were quite sizable, in many cases as much as ten times the process time of the machines. Considering that the traditional model used Machines 12, 13, and 14, as much as 4-6 times the amount of time as the object-oriented model, it is fairly easy

to see how those intercellular travel times would add up and greatly affect the "time in system" statistics. The higher utilization of Machines 12, 13, and 14 is believed to be caused by the nuances of how each model selects from the distributions. The models would sample the distributions differently, directing parts into the system in different orders. The order in which the parts enter the system affects which machines would be available at any given time. While, over time, both models would produce the same results in terms of part mix produced, the rules of the traditional model seem to select parts, and their process plans, in such a way that the alternate machines in Cell 3 are used more often than in the object-oriented model.

Another possible explanation for the difference in "time in system" statistics centres on the fact that in Silk, the user determines how the statistic will be measured, while in Witness, the statistic is measured automatically. While all best efforts were made to define the Silk statistic in the same manner that the Witness statistic is defined, given that the Witness statistic is an internal function in a closed software system, it is difficult to know for certain that these statistics are being measured in exactly the same way at exactly the same time. Even a small difference would, over time, add up. Slight differences in statistic definition, high utilization of machines that require significant intercellular travel time, and the underlying fundamental difference in the approach to model development between the two models all provide explanations as to the differences in the "time in system" statistic.

There is another factor, briefly mentioned above, that may cause slightly statistically dissimilar models. The Silk simulation tool is entity-oriented whereas the Witness simulation tool is resource-oriented. That is, in Silk, the entity (part) is the active element, traveling from resource to queue to resource under its own power. The entity is the driving

force, and essentially tries to push its way through the logic. In Witness, the entities (parts) are generally passive, relying on the resources, queues, etc. to move the entity through the system. Control lies with the resources, which decide when a part will be pulled to or pushed from a resource. While both tools can also be used to simulate the opposite case (i.e. Silk simulations can be written that allow the resources to have some control over part movement and Witness simulations can be modeled to have active parts), this is a notable difference in the underlying fundamental approach to developing the simulation model. Both models are set up as a flexible manufacturing system, in which the machines are checked for availability before the part moves through the system. However, the factor of whether the part is driving the model (as in Silk), or whether the part is sitting back waiting to be pulled to a machine (as in Witness) may affect each part's individual process plan, especially when alternate machines are available. It is difficult to say what effect this difference in approach to modeling may have had in selecting each part's flow, which would then affect the statistics measured. While either approach is acceptable and will produce a viable model, this difference in control and decision-making power at each stage of the model's process is thought to have had some impact on creating slightly statistically dissimilar models.

It is important to note that an extremely simple example of one part and two machines, modeled in both pieces of software, produced statistically similar results. It is believed that as the complexity of the example system grew, so did the model's susceptibility to the cumulative effects of the internal differences between the two pieces of software. Obviously as more factors are introduced into a model, as it becomes more complex, randomness and error may take a bigger toll. However, it is not the goal here to produce exactly the same numerical results. The goal is to determine if the models are behaving

similarly enough to determine that they have, in fact, modeled the same system. Several of the measurables produced statistically similar results, with many others falling within what is deemed to be an acceptable range. The "time in system" statistic carried the largest differences, but explanations have been given.

To summarize, the two models produce statistically similar results for many statistics. For those statistics that are different, the values still fall within a reasonable range, and valid explanations are available. Therefore, the results of the two models can be considered sufficiently similar to indicate that the same system is being modeled using the two different approaches.

6.4 Discussion

The following will briefly discuss the benefits and disadvantages of the two pieces of software, compared to the other, in terms of developing the actual model. It is important to note that these benefits and disadvantages are with regards to comparison with the other software only. Any of the benefits or disadvantages of Witness may or may not be available or solved in other simulation software that employs the more traditional programming paradigm. The same is to be said of Silk with regards to other object-oriented software. The purpose of this discussion is to simply compare the procedures of developing models, in terms of ease of development, using these two pieces of software specifically. The discussion is based solely on the experience of the author in developing these two example models.

Silk

As explained previously, developing a simulation in Silk requires coding objects, and compiling this code to run the simulation. In the case of the above example, twelve class files were written to be compiled, including ten part family files, a simulation file and a file

for the distinguished class called *FMSMfgCell*. The process described in Chapter 5 (and specifically for this example in section 6.2), provided the basis for the model. The remaining step was to write the code, which can be found in the appendix. The process of the user developing the code proved to be an advantage. The user is able to directly manipulate the flow of the part through its process plan by using generic coding principles and Java statements, as opposed to searching for Witness rules and functions. This allowed for a more straightforward process description and easier model development. This benefit exists partly because of the entity-oriented structure of Silk. The details of the process plan could be described in terms of the part, that is, the entire process plan for each part could be described in the part family class. In Witness, the resources generally control the part through a series of pushes and pulls. Despite this difference, it is the ability of users to develop their own code that operates at the same level as the existing software structures (an advantage of object-oriented simulation), that increases the ease with which the model is developed. Developing the Silk model (factoring in all aspects of model building and verifying), including working through the process described in Chapter 5, took only about 70% of the time it took to develop the Witness model. If this time difference were translated into cost, it would represent a significant savings in resources. It is, however, important to note that this time savings is based solely on the experience of one user, developing one example. Further tests would need to be run with several independent users and several examples to verify its validity.

Although the development environment used to develop the Java files attempts some error checking through text colour changes, many syntax or typographic errors were not detected until the entire project was compiled. This generally involved a long list of errors at

initial compilation, and time-consuming backtracking to correct the errors. Although this is fairly common in programming using any language, when compared to an included Witness feature, it proved to be a disadvantage. Another disadvantage of Silk involved statistic collection. Silk has a variety of classes, variables, and methods available for collecting any statistic the modeler may desire. However, the user must decide how the statistic is to be measured, and ensure that the proper code is in place to calculate and record such information. Compared to Witness, which automatically reports many important statistics, this is a disadvantage of Silk software, especially for a new user.

Witness

Witness offers users a full-blown graphic user interface through which simulation development is possible. The user can put the model together visually, adding logic and element details through the use of dialog boxes. These dialog boxes automatically check for rule validation and errors in syntax. Thus, any mistakes of this nature are generally identified fairly quickly, eliminating the need for large amounts of debugging to discover simple syntax or typographic errors. The visual modeling, along with the automatic checking capabilities, proved to be advantageous in developing the Witness model, compared to Silk.

A major disadvantage of the Witness software, compared to the Silk software, was noticed when the part routings were coded. In the above example, there are several part families, each following a different process plan. The coding and logic of the part routings became fairly cumbersome in Witness when working with the machine details. Because many of the part routing details were located in individual machine details (each of which served a variety of parts), numerous variables and logic statements were required. Keeping track of progress, in terms of directing parts and completing routes, while detailing the

machines, proved at times to be somewhat tricky. Because the user must work within the predefined graphic user interfaces and dialog boxes, extra time was spent on discovering the Witness functions and rules that would allow for the desired result. This proved to be a disadvantage for Witness that didn't exist in Silk.

In the object-oriented model, the process details are included in the part class. The description of the part's process is fairly straightforward, especially since the user can code the description as s/he likes. In the traditional model, the process plan is described in the machine details. Control is given to the machine, which serves many different part families. The difficulty arises when alternate machines are brought into the picture. The part routing feature, which in other circumstances is very handy for describing different part process plans, does not work as well since there is no option for an alternate machine at each stage. Thus, the process must be controlled through a series of *if* statements, variables, *pushes*, and *pulls* in the machine logic box.

In fact, the actual results of the two models here are not as important as the lessons learned and comparisons made during the building of the models. The object-oriented simulation allowed for easier and faster development of the model. The simulation specific object-oriented analysis technique described in Chapter 5 provided a strong foundation for the coding of the model, allowing the process to run much more smoothly and quickly than it would have otherwise.

Chapter 7: Summary and Conclusions

7.1 Summary and conclusions

Object-oriented simulations are emerging as the new paradigm in simulation modeling. An approach was developed here to examine manufacturing systems to be simulated in terms of objects. The approach was based on the general layout of several popular object-oriented analysis techniques and adapted for the purpose of simulation using existing object-oriented simulation software.

This approach will assist both the new and slightly experienced user in breaking down a given manufacturing system in terms of objects so that an object-oriented simulation can be performed. It allows the objects to be more visible, as well as ensures that no important system components or measurables are forgotten in an environment that does not prompt the user for information. The approach was applied to an example case and that example case was modeled in both an object-oriented and traditional simulation environment. It was determined that the proposed approach allowed for easier development of an object-oriented simulation model since the necessary objects and relationships were clearly identified and understood before the modeling began. The development of the object-oriented model also proved to be faster and easier than the development of the traditional model, in part because the user is able to code the model without being constrained by pre-existing structures. Any aspect of the model that the user codes exists at the same level as the pre-existing classes.

This method is easily extended to other manufacturing systems or other application areas. It is not specific to manufacturing systems, nor was it ever intended to be. The same series of steps should lead to identification of patients and doctors in a hospital, or passengers and luggage in an airport. More importantly, in some cases, the method may need to be

expanded in terms of number of steps. As it exists right now, this method should cover all necessary steps for most examples. However, there are always some exceptions. As the novice object-oriented modeler uses this procedure, missing phases for particular examples should be documented, analyzed for their frequency, and added at the discretion of the modeler. It can also be extended beyond the application to Silk. As only Steps 8 and 10 are specific to Silk, it could also be extended for use with other software systems, if desired. Step 8 could be modified to represent the structure of another piece of software to ensure that all necessary classes in that software have been identified and analyzed. The essence of Step 10 would remain the same, which is to match the objects of the analysis to the software's object classes. If another piece of software were used, the only difference would be the classes to which the objects are matched.

In addition to providing guidance to novice object-oriented simulation analysts and builders, this method expands the application of object-oriented analysis. Instead of being used solely for software development purposes, it now assists in developing object-oriented simulations by extracting the elements of object-oriented analysis that are relevant to the field of simulation.

7.2 Future Work

The object-oriented analysis for simulation approach described here was developed and tested in the context of manufacturing systems. Future work could include using examples from outside the manufacturing world to determine if any additional steps are required that may be specific to other applications. Although the approach was developed using Silk, the approach could be considered somewhat generic. It would also be interesting in future work to investigate the approach's flexibility in adapting to other object-oriented

software languages. The approach as it stands is fairly labour intensive, though the understanding developed through the performance of that labour serves to make the development of the model easier. However, if automation of the approach through some type of computer program was developed, this may increase the user-friendliness and organization of the approach. Because the noun extraction process is used, this would also pose some interesting questions and potential for interactions with the field of natural language processing.

Further testing should also be done to verify the statement that the object-oriented model took roughly 70% of the time to develop as the traditional model. This testing should involve several different examples of various sizes, using several different simulation modelers, preferably those familiar with the software being used. Only once this testing is completed would that claim hold some sort of validity.

Animation and graphics were not considered in this work. Future work could also include the study and incorporation of animation and graphics in this object-oriented simulation environment

This is only some of the work that could continue with the development of this approach. Of course, if future work in object-oriented simulation is to be considered, given the relative youth of this new paradigm, the possibilities are seemingly endless.

Glossary of Terms

Attributes - pieces of data whose values are specific for each object in a class

Class – an abstract data type of some concept/entity that captures all the characteristics and behaviours common to all instances of the class

Inheritance – an object-oriented programming concept by which objects in each class inherit behaviours and actions from the objects and classes above them

Object – unique instances of a class that represent actual entities and have identities, states and behaviours

Object-oriented analysis (OOA) – the area of study/array of techniques that attempt to identify the objects and classes in a given system and/or domain

Object-oriented design (OOD) – the process of designing software in terms of objects

Object-oriented programming (OOP) – an approach to programming that uses objects as building blocks that work together to distribute the control and functionality of the program

Object-oriented simulation (OOS) – a new paradigm in simulation modeling that uses principles of object-oriented programming in an attempt to increase the power and reusability of simulation models

Polymorphism – an object-oriented programming concept that allows the same property to be applied to different objects

Use-Case Modeling – an object-oriented analysis technique that helps to define the typical scenarios encountered by the user

References

[1] Davis, W.J., Looking into the Future of Simulation, IIE Solutions, Vol. 30, No. 5, May 1998 pp. 25-30.

[2] Banks, J., Principles of Simulation, Chapter 1 in *Handbook of Simulation: Principles, Methodology, Advances, Applications and Practice*, (Ed. Jerry Banks), John Wiley & Sons, Toronto, 1998.

[3] Harrell, C. and K. Tumay, Simulation Made Easy, IIE Solutions, Volume 29, no. 7, July 1997, pp. 39-41

[4] Law, A.M. and W.D. Kelton, Simulation Modeling and Analysis, McGraw-Hill Book Company, Toronto, 1982.

[5] Bender, G.G. and K.Y. Chang, Simulating Roadway and Curbside Traffic at Las Vegas McCarran International Airport, IIE Solutions, Vol. 29, No. 11, November 1997, pp. 26-32.

[6] Joines, J.A. and S.D. Roberts, Object-Oriented Simulation, Chapter 11 in *Handbook of Simulation: Principles, Methodology, Advances, Applications and Practice* (Ed. Jerry Banks), John Wiley & Sons, Toronto, 1998.

[7] Joines, J.A., and S.D. Roberts, Design of Object Oriented Simulations in C++, Proceedings of the 1995 Winter Simulation Conference, Arlington, VA, Dec. 3-6, 1995, pp. 82-89.

[8] Narayanan, S., D.A. Bodner, U. Sreekanth, T. Govindarag, L.F. McGinnis, C.M.Mitchell, Research in object-oriented manufacturing simulations: an assessment of the state of the art, IIE Transactions, Vol. 30, 1998, pp. 795-810

[9] Zeigler, B.P., Hierarchical, Modular Discrete-event Modelling in an Object-Eriented environment, Simulation, Vol 49, No. 5, November 1987, pp. 219-230.

[10] Ulgen, O.M., and T. Thomasma, SmartSim: An object-oriented simulation program generator for manufacturing systems, International Journal of Production Research, Vol. 28, No. 9, 1990, pp. 1713-1730.

[11] Jakksi, A., A Method for Your First Object-Oriented Project, Journal of Object Oriented Programming, Vol. 10, No. 8, 1998, pp. 17-25.

[12] Schach, Stephen R., Classical and Object-Oriented Software Engineering 4th ed., McGraw-Hill, Toronto, 1999.

[13] Harrell, Charles, Ph.D., & K. Tumay, Simulation Made Easy, Engineering and Management Press, Norcross, Georgia, 1995.

[14] Pascoe, G.A., Elements of Object-Oriented Programming, BYTE, Vol. 11, August 1986, pp. 139-144.

[15] Riel, Arthur J., Object-Oriented Design Heuristics, Addison-Wesley, Don Mills, Ontario, 1996.

[16] Korson, T. and J.D. McGregor, Understanding OO: A Unifying Paradigm, Communications of the ACM, Vol. 33, No. 9, September 1990 pp. 41-60.

[17] Deitel, H.M. and P.J. Deitel, Java How to Program, 2nd edition, Prentice Hall, New Jersey, 1998.

[18] Wirfs-Brock, R.J. and R.E. Johnson, Surveying Current Research in Object-Oriented Design, Communications of the ACM, Vol. 33, No. 9, September 1990 pp. 104-124.

[19] Jarke, M., Scenarios for Modeling, Communications of the ACM, Vol. 42, No. 1, Jan. 1999, pp. 47-48.

[20] Rowlett, T., Building an Object Process Around Use Cases, Journal of Object Oriented Programming, Vol. 11, No. 1, March/April 1998, pp. 53-58

[21] Brinkkemper, S., S. Hong, A. Bulthuis, G. van den Goor, Object-Oriented Analysis and Design Methods a Comparative Review, January 1995, Available at: http://wwwis.cs.utwente.nl:8080/dmrg/OODOC/oodoc/oo.html

[22] Rumbaugh, J., Modeling through the Years, Journal of Object-Oriented Programming, Vol. 10, No. 4, July/August, 1997, pp. 16-19.

[23] Coad, P. and E. Yourdon, Object Oriented Analysis, 2nd ed., Yourdon Press, Englewood Cliffs, New Jersey, 1991.

[24] Wirfs-Brock, Rebecca, B. Wilkerson, and L. Wiener, Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[25] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, Object Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[26] Booch, G., Object-Oriented Analysis and Design with Applications 2nd ed. , Benjamin Cummings Publishing Company, Inc. Don Mills, Ontario, 1994.

[27] Shlaer, S. and S. J. Mellor, Object Lifecycles Modeling the World in States, Yourdon Press, Englewood Cliffs, New Jersey, 1992.

[28] Martin, J., Principles of Object-Oriented Design and Analysis, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[29] Rational, The E-Development Center, Frequently Asked Questions, 2001, Available at: http://www.rational.com/uml/gstart/faq.jsp

[30] Object Management Group Inc., What Is OMG-UML and Why Is It Important? 2001, Available at: http://cgi.omg.org/news/pr97/umlprimer.html

[31] The Simula Programming Language, 1996
Available at:http://www.engin.umd.umich.edu/CIS/course.des/cis400/simula/simula.html

[32] ASU (Association of Simula Users)
http://www.isima.fr/asu/

[33] Aesop Corporation, Simple++: Optimization of Systems and Business Processes, Brochure.

[34] Wood, B. and K. Tumay, MODSIM III and Caci's Applications
http://www.caci.com

[35] Sawhney, A., J. Manickam, A. Mund and J. Marble, Java-Based Simulation of Construction Processes using Silk, Proceedings of the 1999 Winter Simulation Conference, IEEE, Piscataway, NJ.

[36] Healy, K.J., and R.A. Kilgore, The Future of Java-Based Simulation, Proceedings of the 1998 Winter Simulation Conference, IEEE. Piscataway, NJ, 1998, 1707-1712.

[37] Healy, K.J., and R.A. Kilgore, Introduction to Silk and Java-Based Simulation, Proceedings of the 1998 Winter Simulation Conference, IEEE. Piscataway, NJ, 1998, pp. 327-334.

[38] Alami, D., Evaluating a New Heuristic Scheduling Rule for Cellular Manufacturing Systems via Simulation, M.A.Sc. Thesis, Industrial and Manufacturing Systems Engineering, University of Windsor, 1999.

[39] The Lanner Group, http://www.lanner.com

Appendix

Silk Simulation Code

Comments are in italics and marked by //

FMSMfgCell

```
//Imports required libraries
import com.threadtec.silk.Silk;
import com.sun.java.swing.JApplet;

//Creates an instance of an applet
public class FMSMfgCells extends JApplet {
   Silk mySilk;

   //Creates new applet called FMSMfgCells and starts
   //simulation
   public static void main(String args[]) {
      FMSMfgCells appletSilk = new FMSMfgCells();
      appletSilk.start();
   }

   //Initial Condition is to start applet (simulation)
   public void init() {
      Silk.setApplet (this);
   }

   //Once applet is started, this message is printed
   //out and the simulation begins
   public void start () {
      System.out.println ("\nExecuting FMSMfgCells...\n");
      mySilk = new Silk();
      mySilk.begin();
   }

   //When simulation stops, system is cleaned and
   //simulation object is nullified
   public void stop() {
      com.threadtec.silk.util.CleanUp.purgeAll();
      mySilk=null;
   }
}
```

Simulation

```
//Import necessary libraries
import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;
```

```
//Defines the resources, queues, queue lengths and machine utilization
//statistics.  They are defined here since every part family will use
//them.  By defining them here, they only need to be defined once.
public class Simulation extends Silk {
   public static Resource resmc1 = new Resource ("MC1");
   public static Resource resmc2 = new Resource ("MC2");
   public static Resource resmc3 = new Resource ("MC3");
   public static Resource resmc4 = new Resource ("MC4",2);
   public static Resource resmc5 = new Resource ("MC5",2);
   public static Resource resmc6 = new Resource ("MC6");
   public static Resource resmc7 = new Resource ("MC7");
   public static Resource resmc8 = new Resource ("MC8");
   public static Resource resmc9 = new Resource ("MC9");
   public static Resource resmc10 = new Resource ("MC10",2);
   public static Resource resmc11 = new Resource ("MC11");
   public static Resource resmc12 = new Resource ("MC12");
   public static Resource resmc13 = new Resource ("MC13");
   public static Resource resmc14 = new Resource ("MC14");

   public static Queue quemc1 = new Queue ("qmc1");
   public static Queue quemc2 = new Queue ("qmc2");
   public static Queue quemc3 = new Queue ("qmc3");
   public static Queue quemc4 = new Queue ("qmc4");
   public static Queue quemc5 = new Queue ("qmc5");
   public static Queue quemc6 = new Queue ("qmc6");
   public static Queue quemc7 = new Queue ("qmc7");
   public static Queue quemc8 = new Queue ("qmc8");
   public static Queue quemc9 = new Queue ("qmc9");
   public static Queue quemc10 = new Queue ("qmc10");
   public static Queue quemc11 = new Queue ("qmc11");
   public static Queue quemc12 = new Queue ("qmc12");
   public static Queue quemc13 = new Queue ("qmc13");
   public static Queue quemc14 = new Queue ("qmc14");

   public static Queue fmsqueue1 = new Queue ("fmsq1");
   public static Queue fmsqueue2 = new Queue ("fmsq2");
   public static Queue fmsqueue3 = new Queue ("fmsq3");
   public static Queue fmsqueue4 = new Queue ("fmsq4");
   public static Queue fmsqueue5 = new Queue ("fmsq5");
   public static Queue fmsqueue6 = new Queue ("fmsq6");
   public static Queue fmsqueue7 = new Queue ("fmsq7");
   public static Queue fmsqueue8 = new Queue ("fmsq8");
   public static Queue fmsqueue9 = new Queue ("fmsq9");
   public static Queue fmsqueue10 = new Queue ("fmsq10");
```

```
  public static TimeDependent tdsmc1QLength = new TimeDependent (quemc1.length,
"MC1 Queue length");
  public static TimeDependent tdsmc2QLength = new TimeDependent (quemc2.length,
"MC2 Queue length");
  public static TimeDependent tdsmc3QLength = new TimeDependent (quemc3.length,
"MC3 Queue length");
  public static TimeDependent tdsmc4QLength = new TimeDependent (quemc4.length,
"MC4 Queue length");
  public static TimeDependent tdsmc5QLength = new TimeDependent (quemc5.length,
"MC5 Queue length");
  public static TimeDependent tdsmc6QLength = new TimeDependent (quemc6.length,
"MC6 Queue length");
  public static TimeDependent tdsmc7QLength = new TimeDependent (quemc7.length,
"MC7 Queue length");
  public static TimeDependent tdsmc8QLength = new TimeDependent (quemc8.length,
"MC8 Queue length");
  public static TimeDependent tdsmc9QLength = new TimeDependent (quemc9.length,
"MC9 Queue length");
  public static TimeDependent tdsmc10QLength = new TimeDependent (quemc10.length,
"MC10 Queue length");
  public static TimeDependent tdsmc11QLength = new TimeDependent (quemc11.length,
"MC11 Queue length");
  public static TimeDependent tdsmc12QLength = new TimeDependent (quemc12.length,
"MC12 Queue length");
  public static TimeDependent tdsmc13QLength = new TimeDependent (quemc13.length,
"MC13 Queue length");
  public static TimeDependent tdsmc14QLength = new TimeDependent (quemc14.length,
"MC14 Queue length");

  public static TimeDependent tdsMC1Util = new TimeDependent (resmc1.numBusy,
"MC1 utilization");
  public static TimeDependent tdsMC2Util = new TimeDependent (resmc2.numBusy,
"MC2 utilization");
  public static TimeDependent tdsMC3Util = new TimeDependent (resmc3.numBusy,
"MC3 utilization");
  public static TimeDependent tdsMC4Util = new TimeDependent (resmc4.numBusy,
"MC4 utilization");
  public static TimeDependent tdsMC5Util = new TimeDependent (resmc5.numBusy,
"MC5 utilization");
  public static TimeDependent tdsMC6Util = new TimeDependent (resmc6.numBusy,
"MC6 utilization");
  public static TimeDependent tdsMC7Util = new TimeDependent (resmc7.numBusy,
"MC7 utilization");
  public static TimeDependent tdsMC8Util = new TimeDependent (resmc8.numBusy,
"MC8 utilization");
```

```
  public static TimeDependent tdsMC9Util = new TimeDependent (resmc9.numBusy,
"MC9 utilization");
  public static TimeDependent tdsMC10Util = new TimeDependent (resmc10.numBusy,
"MC10 utilization");
  public static TimeDependent tdsMC11Util = new TimeDependent (resmc11.numBusy,
"MC11 utilization");
  public static TimeDependent tdsMC12Util = new TimeDependent (resmc12.numBusy,
"MC12 utilization");
  public static TimeDependent tdsMC13Util = new TimeDependent (resmc13.numBusy,
"MC13 utilization");
  public static TimeDependent tdsMC14Util = new TimeDependent (resmc14.numBusy,
"MC14 utilization");

  public static Uniform unitest = new Uniform (1,100,22350);
  // Sets initial conditions
  public void init () {


  //Defines each part family as an Entity and as a class.
  //Defines each family's start time as time 0.
  FMSPart1 entFMSPart1;
  entFMSPart1= (FMSPart1)newEntity(FMSPart1.class);
  entFMSPart1.start(0.0);

  FMSPart2 entFMSPart2;
  entFMSPart2= (FMSPart2)newEntity(FMSPart2.class);
  entFMSPart2.start(0.0);

  FMSPart3 entFMSPart3;
  entFMSPart3= (FMSPart3)newEntity(FMSPart3.class);
  entFMSPart3.start(0.0);

  FMSPart4 entFMSPart4;
  entFMSPart4= (FMSPart4)newEntity(FMSPart4.class);
  entFMSPart4.start(0.0);

  FMSPart5 entFMSPart5;
  entFMSPart5= (FMSPart5)newEntity(FMSPart5.class);
  entFMSPart5.start(0.0);

  FMSPart6 entFMSPart6;
 entFMSPart6= (FMSPart6)newEntity(FMSPart6.class);
 entFMSPart6.start(0.0);

  FMSPart7 entFMSPart7;
  entFMSPart7= (FMSPart7)newEntity(FMSPart7.class);
```

```
   entFMSPart7.start(0.0);

   FMSPart8 entFMSPart8;
   entFMSPart8= (FMSPart8)newEntity(FMSPart8.class);
   entFMSPart8.start(0.0);

   FMSPart9 entFMSPart9;
   entFMSPart9= (FMSPart9)newEntity(FMSPart9.class);
   entFMSPart9.start(0.0);

   FMSPart10 entFMSPart10;
   entFMSPart10= (FMSPart10)newEntity(FMSPart10.class);
   entFMSPart10.start(0.0);

   }
   //Specifications for running simulation
   public void run() {

   //Runs 50 replications of 9600 time units each
   setReplications(50);
   setRunLength(9600.);
   }
}
```

Part 1

```
//Import the necessary libraries into the program

import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
 public class FMSPart1 extends Entity {

   //Defines the variables to be used in the program.  attArrivalTime is
   //used to calculate time in system; test1 is used in the distribution
   //that determines part mix; numpart1 is used to count the number of
   //parts entering the system
   double attArrivalTime;
   double test1;
   int numpart1=0;


   //Defines the statistics for time in system and number of parts (entering) statistics
   //as well as the distribution for the part mix.
```

```
static Observational obsTimeInSystem1 = new Observational ("Time in system1");
static Uniform unitest1 = new Uniform (1,100,43663);
static Observational obsnumpart1 = new Observational ("Number of Part 1 entered");
static Observational obstest1 = new Observational ("Test 1 values");

// Defines the methods.
// Begins with required "process"

public void process ()
{


    //Creates the entity. Tests the distribution to determine
    //what part type the part will be (to adjust for orders). If it falls within Part 1 range, it
waits in
    //FMSqueue1 until the proper machine is available. Once the part enters the system,
time is assigned to
    //the Arrival time attribute. Counts the part entering the system. Part begins its process
via a series of
    //seize, delay and release commands.

    create(15);
    test1=unitest1.sample();
    obstest1.record(test1);
    if (test1<=8) {
      queue (fmsqueue1);
     if((resmc1.availability.value!=0) || (resmc13.availability.value!=0)){
   //Uses the alternate machine. If the first resource is busy
   //the part is routed to the alternate machine
     if (resmc1.availability.value==0){
        seize(resmc13);
        attArrivalTime=time;
        numpart1= 1;
        obsnumpart1.record(numpart1);
        dequeue (fmsqueue1);
        delay (3.6);
        release (resmc13);
        delay(10);
     }
     else {
   //Seizes the resource and so empties part from queue.
   //Calculates time in queue. Delays the part in the resource
   //for the required amount of time.

        seize (resmc1);
        attArrivalTime=time;
```

```
    numpart1= 1;
    obsnumpart1.record(numpart1);
    dequeue (fmsqueue1);
    delay (1.6);
    release (resmc1);
}


queue (quemc3);
while (condition (resmc3.getAvailability()==0));
seize (resmc3);
dequeue (quemc3);
delay (0.8);
release (resmc3);

queue (quemc5);
if (resmc5.availability.value==0){
   dequeue (quemc5);
   delay(10);
   queue (quemc12);
   while (resmc12.availability.value==0);
   seize (resmc12);
   dequeue (quemc12);
   delay (2.1);
   release (resmc12);
   delay (10);}
else {
   seize (resmc5);
   dequeue (quemc5);
   delay (4.1);
   release (resmc5);
}

queue (quemc4);

if (resmc4.availability.value==0){
   dequeue (quemc4);
   delay(10);
   queue (quemc6);
   while  (resmc6.availability.value==0);
   seize (resmc6);
   dequeue (quemc6);
   delay (1.4);
   release (resmc6);
}
else {
```

```
            seize (resmc4);
            dequeue (quemc4);
            delay (1.4);
            release (resmc4);
          }


          //Records time in system and disposes of part
          obsTimeInSystem1.record(time-attArrivalTime);


        }
        } dispose ();
      }

      // If the part that was created above does not pass the
      //statistical distribution test, it is disposed of immediately

  }
```

Part 2

```
//Import the necessary libraries into the program

import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
public class FMSPart2 extends Entity {

  //Defines the variables to be used in the program.  attArrivalTime is
   //used to calculate time in system; test2 is used in the distribution
   //that determines part mix; numpart2 is used to count the number of
   //parts entering the system

   double attArrivalTime;
   double test2;
   int numpart2=0;


   //Defines the statistics for time in system and number of parts (entering) statistics
   //as well as the distribution for the part mix.
```

```
static Observational obsTimeInSystem2 = new Observational ("Time in system2");
static Uniform unitest2 = new Uniform (1,100,58226);
static Observational obsnumpart2 = new Observational ("Number of Part 2 entered");
static Observational obstest2 = new Observational ("Test 2 values");

// Defines the methods.
// Begins with required "process"
public void process ()
{
  //Creates the entity. Tests the distribution to determine
  //what part type the part will be (to adjust for orders). If it falls within Part 2 range, it
waits in
  //FMSqueue2 until the proper machine is available. Once the part enters the system,
time is assigned to
  //the Arrival time attribute. Counts the part entering the system. Part begins its process
via a series of
  //seize, delay and release commands.

  create(15);
  test2=unitest2.sample();
  obstest2.record(test2);

  if (test2>8 && test2 <= 35) {
    queue (fmsqueue2);
    if((resmc2.availability.value!=0)||(resmc3.availability.value!=0)){
  //Uses the alternate machine. If the first resource is busy
  //the part is routed to the alternate machine
    if (resmc2.availability.value==0){
      seize (resmc3);
      attArrivalTime=time;
      numpart2=1;
      obsnumpart2.record(numpart2);
      dequeue (fmsqueue2);
      delay (3.8);
      release (resmc3);
    }
    else {
      seize (resmc2);
      attArrivalTime=time;
      numpart2=1;
      obsnumpart2.record(numpart2);
      dequeue (fmsqueue2);
      delay (0.8);
      release (resmc2);
    }
```

```
//Seizes the resource and so empties part from queue.
//Calculates time in queue.  Delays the part in the resource
//for the required amount of time.

queue (quemc1);
if (resmc1.availability.value==0){
   dequeue (quemc1);
   delay (10);
   queue (quemc13);
   while (condition (resmc13.getAvailability()==0));
   seize (resmc13);
   dequeue (quemc13);
   delay (2.6);
   release (resmc13);
   delay (10);
}
else {
   seize (resmc1);
   dequeue (quemc1);
   delay (0.6);
   release (resmc1);
}

queue (quemc5);
if (resmc5.availability.value==0){
   dequeue (quemc5);
   delay (10);
   queue (quemc12);
   while (condition (resmc12.getAvailability()==0));
   seize (resmc12);
   dequeue (quemc12);
   delay (2.6);
   release (resmc12);
   delay (10);
}
else {
   seize (resmc5);
   dequeue (quemc5);
   delay (0.6);
   release (resmc5);
}

queue (quemc4);
if (resmc4.availability.value==0){
   dequeue (quemc4);
   delay (10);
```

```
        queue (quemc6);
        while (condition (resmc6.getAvailability()==0));
        seize (resmc6);
        dequeue (quemc6);
        delay (4.0);
        release (resmc6);
      }
      else {
        seize (resmc4);
        dequeue (quemc4);
        delay (1.0);
        release (resmc4);
      }


        //Records time in system and disposes of part
        obsTimeInSystem2.record(time-attArrivalTime);


      }
      }

        dispose();

      }
      // If the part that was created above does not pass the
      //statistical distribution test, it is disposed of immediately

  }
```

Part 3

```
//Import the necessary libraries into the program

import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
public class FMSPart3 extends Entity {

  //Defines the variables to be used in the program.  attArrivalTime is
  //used to calculate time in system; test3 is used in the distribution
  //that determines part mix; numpart3 is used to count the number of
  //parts entering the system
```

```
double attArrivalTime;
double test3;
int numpart3;

//Defines the statistics for time in system and number of parts (entering) statistics
//as well as the distribution for the part mix.

static Observational obsTimeInSystem3 = new Observational ("Time in system3");
static Uniform unitest3 = new Uniform (1,100,98108);
static Observational obsnumpart3 = new Observational ("Number of Part 3 entered");
static Observational obstest3 = new Observational ("Test 3 values");

// Defines the methods.
// Begins with required "process"
public void process ()
{
  //Creates the entity. Tests the distribution to determine
  //what part type the part will be (to adjust for orders). If it falls within Part 3 range, it
waits in
  //FMSqueue3 until the proper machine is available. Once the part enters the system,
time is assigned to
  //the Arrival time attribute. Counts the part entering the system. Part begins its process
via a series of
  //seize, delay and release commands.

  create(15);
  test3=unitest3.sample();
  obstest3.record(test3);

  if (test3>35 && test3<=43) {
    queue (fmsqueue3);

  //Uses the alternate machine. If the first resource is busy
  //the part is routed to the alternate machine
    if(resmc6.availability.value!=0){
     dequeue(fmsqueue3);
     queue(quemc6);
     seize (resmc6);
     attArrivalTime=time;
     numpart3=1;
     obsnumpart3.record(numpart3);
     dequeue (quemc6);
     delay (1.4);
     release (resmc6);

  //Seizes the resource and so empties part from queue.
```

```
//Calculates time in queue.  Delays the part in the resource
//for the required amount of time.

   queue (quemc10);
   while (condition (resmc10.getAvailability()==0));
   seize (resmc10);
   dequeue (quemc10);
   delay (1.0);
   release (resmc10);

   queue (quemc9);
   while (condition (resmc9.getAvailability()==0));
   seize (resmc9);
   dequeue (quemc9);
   delay (0.4);
   release (resmc9);

   queue (quemc8);

   if (resmc8.availability.value==0){
      dequeue (quemc8);
      queue (quemc11);
      while (condition (resmc11.getAvailability()==0));
      seize (resmc11);
      dequeue (quemc11);
      delay (0.6);
      release (resmc11);
   }
   else {
      seize (resmc8);
      dequeue (quemc8);
      delay (0.8);
      release (resmc8);
   }


   //Records time in system and disposes of part
   obsTimeInSystem3.record(time-attArrivalTime);


}
 }

   dispose();
}
// If the part that was created above does not pass the
```

```
    //statistical distribution test, it is disposed of immediately

  }
```

Part 4

```
//Import the necessary libraries into the program

import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
public class FMSPart4 extends Entity {

  //Defines the variables to be used in the program.  attArrivalTime is
  //used to calculate time in system; test4 is used in the distribution
  //that determines part mix; numpart4 is used to count the number of
  //parts entering the system

  double attArrivalTime;
  double test4;
  int numpart4;

  //Defines the time in system and number of parts (entering) statistics
  //as well as the distribution for the part mix.
  static Observational obsTimeInSystem4 = new Observational ("Time in system4");
  static Uniform unitest4 = new Uniform (1,100,53754);
  static Observational obsnumpart4 = new Observational ("Number of Part 4 entered");
  static Observational obstest4 = new Observational ("Test 4 values");

  // Defines the methods.
  // Begins with required "process"
  public void process ()
  {
    //Creates the entity.  Tests the distribution to determine
    //what part type the part will be (to adjust for orders).  If it falls within Part 4 range, it
waits in
    //FMSqueue4 until the proper machine is available.  Once the part enters the system,
time is assigned to
    //the Arrival time attribute.  Counts the part entering the system.  Part begins its process
via a series of
    //seize, delay and release commands.

    create(15);
    test4=unitest4.sample();
```

```
obstest4.record(test4);

if (test4> 43 && test4<=49) {
  queue (fmsqueue4);
  //Uses the alternate machine.  If the first resource is busy
  //the part is routed to the alternate machine

  if (resmc3.availability.value!=0){
   dequeue(fmsqueue4);
   queue (quemc3);
   seize (resmc3);
   attArrivalTime=time;
   numpart4=1;
   obsnumpart4.record(numpart4);
   dequeue (quemc3);
   delay (1.6);
   release (resmc3);

//Seizes the resource and so empties part from queue.
//Calculates time in queue.  Delays the part in the resource
//for the required amount of time.

   queue (quemc1);
   if (resmc1.availability.value==0){
    dequeue (quemc1);
    delay (10);
    queue (quemc13);
    while (condition (resmc13.getAvailability()==0));
    seize (resmc13);
    dequeue (quemc13);
    delay (2.8);
    release (resmc13);
    delay (10);
   }
   else {
    seize (resmc1);
    dequeue (quemc1);
    delay (0.8);
    release (resmc1);
   }

queue (quemc5);
while (condition (resmc5.getAvailability()==0));
seize (resmc5);
dequeue (quemc5);
delay (2.4);
```

```
release (resmc5);

queue (quemc4);

if (resmc4.availability.value==0){
    dequeue (quemc4);
    delay (10);
    queue (quemc6);
    while (condition (resmc6.getAvailability()==0));
    seize (resmc6);
    dequeue (quemc6);
    delay (1.4);
    release (resmc6);
    delay (10);}
 else {seize (resmc4);
      dequeue (quemc4);
      delay (1.4);
      release (resmc4);};

 delay (10);
 queue (quemc14);

 while (condition (resmc14.getAvailability()==0));
 seize (resmc14);
 dequeue (quemc14);
 delay (0.4);
 release (resmc14);

 queue (quemc12);

 while (condition (resmc12.getAvailability()==0));
 seize (resmc12);
 dequeue (quemc12);
 delay (3.0);
 release (resmc12);


 //Records time in system and disposes of part
 obsTimeInSystem4.record(time-attArrivalTime);


 }
  }

    dispose();
}
```

```
    // If the part that was created above does not pass the
    //statistical distribution test, it is disposed of immediately

  }
```

Part 5

```
//Import the necessary libraries into the program

import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
public class FMSPart5 extends Entity {

   //Defines the variables to be used in the program.  attArrivalTime is
   //used to calculate time in system; test5 is used in the distribution
   //that determines part mix; numpart5 is used to count the number of
   //parts entering the system

   double attArrivalTime;
   double test5;
   int numpart5;

   //Defines the time in system and number of parts (entering) statistics
   //as well as the distribution for the part mix.

   static Observational obsTimeInSystem5 = new Observational ("Time in system5");
   static Uniform unitest5 = new Uniform (1,100,30951);
   static Observational obsnumpart5 = new Observational ("Number of Part 5 entered");
   static Observational obstest5 = new Observational ("Test 5 values");

   // Defines the methods.
   // Begins with required "process"
   public void process ()
   {
      //Creates the entity.  Tests the distribution to determine
      //what part type the part will be (to adjust for orders).  If it falls within Part 5 range, it
waits in
      //FMSqueue5 until the proper machine is available.  Once the part enters the system,
time is assigned to
      //the Arrival time attribute.  Counts the part entering the system.  Part begins its process
via a series of
      //seize, delay and release commands.
```

```
create(15);
test5=unitest5.sample();
obstest5.record(test5);

if (test5 >49 && test5<=77) {

  queue (fmsqueue5);
  //Uses the alternate machine.  If the first resource is busy
  //the part is routed to the alternate machine

  if((resmc6.availability.value!=0)||(resmc9.availability.value!=0)){
   if (resmc6.availability.value==0){
       seize (resmc9);
       attArrivalTime=time;
       numpart5=1;
       obsnumpart5.record(numpart5);
       dequeue (fmsqueue5);
       delay (2.2);
       release (resmc9);
    }
   else {
       seize (resmc6);
       attArrivalTime=time;
       numpart5=1;
       obsnumpart5.record(numpart5);
       dequeue (fmsqueue5);
       delay (1.2);
       release (resmc6);
   }

 //Seizes the resource and so empties part from queue.
 //Calculates time in queue.  Delays the part in the resource
 //for the required amount of time.

queue (quemc8);

if (resmc8.availability.value==0){
    dequeue (quemc8);
    queue (quemc11);
    while (condition (resmc11.getAvailability()==0));
    seize (resmc11);
    dequeue (quemc11);
    delay (0.8);
    release (resmc11);}
 else {seize (resmc8);
      dequeue (quemc8);
```

```
        delay (1.8);
        release (resmc8);};

    queue (quemc10);
    while (condition (resmc10.getAvailability()==0));
    seize (resmc10);
    dequeue (quemc10);
    delay (0.8);
    release (resmc10);

     queue (quemc7);

    if (resmc7.availability.value==0){
       dequeue (quemc7);
       delay (10);
       queue (quemc14);
       while (condition (resmc14.getAvailability()==0));
       seize (resmc14);
       dequeue (quemc14);
       delay (2.0);
       release (resmc14);}
    else {seize (resmc7);
        dequeue (quemc7);
        delay (1.0);
        release (resmc7);};


    //Records time in system and disposes of part
    obsTimeInSystem5.record(time-attArrivalTime);


    }
    }

       dispose();
  }
    // If the part that was created above does not pass the
    //statistical distribution test, it is disposed of immediately

  }
```

Part 6

```
//Import the necessary libraries into the program
```

```
import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
public class FMSPart6 extends Entity {

   //Defines the variables to be used in the program.  attArrivalTime is
   //used to calculate time in system; test6 is used in the distribution
   //that determines part mix; numpart6 is used to count the number of
   //parts entering the system

   double attArrivalTime;
   double test6;
   int numpart6;

   //Defines the time in system and number of parts (entering) statistics
   //as well as the distribution for the part mix.

   static Observational obsTimeInSystem6 = new Observational ("Time in system6");
   static Uniform unitest6 = new Uniform (1,100,47862);
   static Observational obsnumpart6 = new Observational ("Number of Part 6 entered");
   static Observational obstest6 = new Observational ("Test 6 values");

   // Defines the methods.
   // Begins with required "process"
   public void process ()
   {
      //Creates the entity.  Tests the distribution to determine
      //what part type the part will be (to adjust for orders).  If it falls within Part 6 range, it
waits in
      //FMSqueue6 until the proper machine is available.  Once the part enters the system,
time is assigned to
      //the Arrival time attribute.  Counts the part entering the system.  Part begins its process
via a series of
      //seize, delay and release commands.

      create(15);
      test6=unitest6.sample();
      obstest6.record(test6);

      if (test6>77 && test6 <= 82) {
         queue (fmsqueue6);
        if ((resmc2.availability.value!=0)||(resmc3.availability.value!=0)){

       //Uses the alternate machine.  If the first resource is busy
```

```
//the part is routed to the alternate machine

  if (resmc2.availability.value==0){
     seize (resmc3);
     attArrivalTime=time;
     numpart6=1;
     obsnumpart6.record(numpart6);
     dequeue (fmsqueue6);
     delay (4.2);
     release (resmc3);
  }
  else {
     seize (resmc2);
     attArrivalTime=time;
     numpart6=1;
     obsnumpart6.record(numpart6);
     dequeue (fmsqueue6);
     delay (1.2);
     release (resmc2);

  }

//Seizes the resource and so empties part from queue.
//Calculates time in queue.  Delays the part in the resource
//for the required amount of time.

queue (quemc4);

if (resmc4.availability.value==0){
   dequeue (quemc4);
   delay (10);
   queue (quemc6);
   while (condition (resmc6.getAvailability()==0));
   seize (resmc6);
   dequeue (quemc6);
   delay (1.4);
   release (resmc6);
   delay (10);}
 else {seize (resmc4);
     dequeue (quemc4);
     delay (1.4);
     release (resmc4);};

 queue (quemc5);
 while (condition (resmc5.getAvailability()==0));
 seize (resmc5);
```

```
    dequeue (quemc5);
    delay (0.8);
    release (resmc5);

    delay (10);
    queue (quemc13);
    while (condition (resmc13.getAvailability()==0));
    seize (resmc13);
    dequeue (quemc13);
    delay (2.0);
    release (resmc13);

    queue (quemc12);

    while (condition (resmc12.getAvailability()==0));
    seize (resmc12);
    dequeue (quemc12);
    delay (1.8);
    release (resmc12);


    //Records time in system and disposes of part
    obsTimeInSystem6.record(time-attArrivalTime);

  }
   }
   dispose();
  }
   // If the part that was created above does not pass the
   //statistical distribution test, it is disposed of immediately

 }
```

Part 7

```
//Import the necessary libraries into the program

import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
public class FMSPart7 extends Entity {

  //Defines the variables to be used in the program.  attArrivalTime is
  //used to calculate time in system; test7 is used in the distribution
```

```
//that determines part mix; numpart7 is used to count the number of
//parts entering the system

double attArrivalTime;
double test7;
int numpart7;

//Defines the time in system and number of parts (entering) statistics
//as well as the distribution for the part mix.

static Observational obsTimeInSystem7 = new Observational ("Time in system7");
static Uniform unitest7 = new Uniform (1,100,96359);
static Observational obsnumpart7 = new Observational ("Number of Part 7 entered");
static Observational obstest7 = new Observational ("Test 7 values");

// Defines the methods.
// Begins with required "process"
public void process ()
{
  //Creates the entity. Tests the distribution to determine
  //what part type the part will be (to adjust for orders). If it falls within Part 7 range, it
waits in
  //FMSqueue7 until the proper machine is available. Once the part enters the system,
time is assigned to
  //the Arrival time attribute. Counts the part entering the system. Part begins its process
via a series of
  //seize, delay and release commands.


   create(15);
   test7=unitest7.sample();
   test7=unitest.sample();
   obstest7.record(test7);
   if (test7>82 && test7<=86) {
    queue (fmsqueue7);
    //Uses the alternate machine. If the first resource is busy
    //the part is routed to the alternate machine
    if (resmc14.availability.value!=0){
    dequeue(fmsqueue7);
    queue (quemc14);
    seize (resmc14);
    attArrivalTime=time;
    numpart7=1;
    obsnumpart7.record(numpart7);
    dequeue (quemc14);
    delay (1.6);
```

```
release (resmc14);

//Seizes the resource and so empties part from queue.
//Calculates time in queue.  Delays the part in the resource
//for the required amount of time.

delay (10);
queue (quemc9);
while (condition (resmc9.getAvailability()==0));
seize (resmc9);
dequeue (quemc9);
delay (2.6);
release (resmc9);
queue (quemc8);

if (resmc8.availability.value==0){
  dequeue (quemc8);
  queue (quemc11);
  while (condition (resmc11.getAvailability()==0));
  seize (resmc11);
  dequeue (quemc11);
  delay (0.8);
  release (resmc11);
}
else {
  seize (resmc8);
  dequeue (quemc8);
  delay (1.0);
  release (resmc8);
}

queue (quemc7);
while (condition (resmc7.getAvailability()==0));
seize (resmc7);
dequeue (quemc7);
delay (1.4);
release (resmc7);

//Records time in system and disposes of part
obsTimeInSystem7.record(time-attArrivalTime);


}
}
dispose();
```

```
    // If the part that was created above does not pass the
    //statistical distribution test, it is disposed of immediately

  }
}
```

Part 8

```
//Import the necessary libraries into the program

import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
public class FMSPart8 extends Entity {

  //Defines the variables to be used in the program.  attArrivalTime is
  //used to calculate time in system; test8 is used in the distribution
  //that determines part mix; numpart8 is used to count the number of
  //parts entering the system

  double attArrivalTime;
  double test8;
  int numpart8;

  //Defines the time in system and number of parts (entering) statistics
  //as well as the distribution for the part mix.
  static Observational obsTimeInSystem8 = new Observational ("Time in system8");
  static Uniform unitest8 = new Uniform (1,100,48953);
  static Observational obsnumpart8 = new Observational ("Number of Part 8 entered");
  static Observational obstest8 = new Observational ("Test 8 values");

  // Defines the methods.
  // Begins with required "process"
  public void process ()
  {
    //Creates the entity.  Tests the distribution to determine
    //what part type the part will be (to adjust for orders).  If it falls within Part 8 range, it
waits in
    //FMSqueue8 until the proper machine is available.  Once the part enters the system,
time is assigned to
    //the Arrival time attribute.  Counts the part entering the system.  Part begins its process
via a series of
    //seize, delay and release commands.
```

```
create (15);
test8=unitest8.sample();
obstest8.record(test8);

if (test8>86 && test8<=91) {
   queue (fmsqueue8);
  if ((resmc1.availability.value!=0)|| (resmc13.availability.value!=0)){

  //Uses the alternate machine.  If the first resource is busy
  //the part is routed to the alternate machine

   if (resmc1.availability.value==0){
      seize (resmc13);
      attArrivalTime=time;
      numpart8=1;
      obsnumpart8.record(numpart8);
      dequeue (fmsqueue8);
      delay (4.8);
      release (resmc13);
      delay (10);}
   else {seize (resmc1);
      attArrivalTime=time;
      numpart8=1;
      obsnumpart8.record(numpart8);
      dequeue (fmsqueue8);
      delay (2.8);
      release (resmc1);

   }

//Seizes the resource and so empties part from queue.
//Calculates time in queue.  Delays the part in the resource
//for the required amount of time.

   queue (quemc5);

   if (resmc5.availability.value==0){
      dequeue (quemc5);
      delay (10);
      queue (quemc12);
      while (condition (resmc12.getAvailability()==0));
      seize (resmc12);
      dequeue (quemc12);
      delay (2.8);
      release (resmc12);
      delay (10);}
```

```
        else {seize (resmc5);
           dequeue (quemc5);
           delay (0.8);
           release (resmc5);
        }

        queue (quemc3);

        while (condition (resmc3.getAvailability()==0));
        seize (resmc3);
        dequeue (quemc3);
        delay (1.2);
        release (resmc3);

        queue (quemc2);
        while (condition (resmc2.getAvailability()==0));
        seize (resmc2);
        dequeue (quemc2);
        delay (2.4);
        release (resmc2);

        delay (10);
        queue (quemc11);

        while (condition (resmc11.getAvailability()==0));
        seize (resmc11);
        dequeue (quemc11);
        delay (1.6);
        release (resmc11);


        //Records time in system and disposes of part
        obsTimeInSystem8.record(time-attArrivalTime);


        }
    }
        dispose();

    }
    // If the part that was created above does not pass the
    //statistical distribution test, it is disposed of immediately

}
```

Part 9

```
//Import the necessary libraries into the program

import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
public class FMSPart9 extends Entity {

   //Defines the variables to be used in the program.  attArrivalTime is
   //used to calculate time in system; test9 is used in the distribution
   //that determines part mix; numpart9 is used to count the number of
   //parts entering the system

   double attArrivalTime;
   double test9;
   int numpart9;


   //Defines the time in system and number of parts (entering) statistics
   //as well as the distribution for the part mix.
   static Observational obsTimeInSystem9 = new Observational ("Time in system9");
   static Uniform unitest9 = new Uniform (1,100,33854);
   static Observational obsnumpart9 = new Observational ("Number of Part 9 entered");
   static Observational obstest9 = new Observational ("Test 9 values");


   // Defines the methods.
   // Begins with required "process"
   public void process ()
   {
     //Creates the entity.  Tests the distribution to determine
     //what part type the part will be (to adjust for orders).  If it falls within Part 9 range, it
waits in
     //FMSqueue9 until the proper machine is available.  Once the part enters the system,
time is assigned to
     //the Arrival time attribute.  Counts the part entering the system.  Part begins its process
via a series of
     //seize, delay and release commands.

      create (15);
      test9=unitest9.sample();
      obstest9.record(test9);
```

```
if (test9>91 && test9<=97) {
   queue (fmsqueue9);

   //Uses the alternate machine.  If the first resource is busy
   //the part is routed to the alternate machine
   if (resmc7.availability.value!=0){
   dequeue(fmsqueue9);
   queue(quemc7);
   seize (resmc7);
   attArrivalTime=time;
   numpart9=1;
   obsnumpart9.record(numpart9);
   dequeue (quemc7);
   delay (3.2);
   release (resmc7);
   queue (quemc8);
   if (resmc8.availability.value==0){
      dequeue (quemc8);
      queue (quemc11);
      while (condition (resmc11.getAvailability()==0));
      seize (resmc11);
      dequeue (quemc11);
      delay (0.8);
      release (resmc11);
   }
   else {
      seize (resmc8);
      dequeue (quemc8);
      delay (1.0);
      release (resmc8);
   }

   //Seizes the resource and so empties part from queue.
   //Calculates time in queue.  Delays the part in the resource
   //for the required amount of time.

   queue (quemc10);
   while (condition (resmc10.getAvailability()==0));
   seize (resmc10);
   dequeue (quemc10);
   delay (0.4);
   release (resmc10);

   delay(10);
   queue (quemc14);
   while (condition (resmc14.getAvailability()==0));
```

```
        seize (resmc14);
        dequeue (quemc14);
        delay (2.8);
        release (resmc14);

        //Records time in system and disposes of part
        obsTimeInSystem9.record(time-attArrivalTime);


      }
      }
       dispose();

    // If the part that was created above does not pass the
    //statistical distribution test, it is disposed of immediately

    }
}
```

Part 10

```
//Import the necessary libraries into the program

import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

//Defines the class
public class FMSPart10 extends Entity {

  //Defines the variables to be used in the program.  attArrivalTime is
  //used to calculate time in system; test10 is used in the distribution
  //that determines part mix; numpart10 is used to count the number of
  //parts entering the system

  double attArrivalTime;
  double test10;
  int numpart10;

  //Defines the time in system and number of parts (entering) statistics
  //as well as the distribution for the part mix.
  static Observational obsTimeInSystem10 = new Observational ("Time in system10");
  static Uniform unitest10 = new Uniform (1,100,92388);
  static Observational obsnumpart10 = new Observational ("Number of Part 10 entered");
  static Observational obstest10 = new Observational ("Test 10 values");
```

```
// Defines the methods.
// Begins with required "process"
public void process ()
{
  //Creates the entity. Tests the distribution to determine
  //what part type the part will be (to adjust for orders). If it falls within Part 10 range, it
waits in
  //FMSqueue10 until the proper machine is available. Once the part enters the system,
time is assigned to
  //the Arrival time attribute. Counts the part entering the system. Part begins its process
via a series of
  //seize, delay and release commands.

  create(15);
  test10=unitest10.sample();
  obstest10.record(test10);
  if (test10>97 && test10<=100) {
    queue (fmsqueue10);

    //Uses the alternate machine. If the first resource is busy
    //the part is routed to the alternate machine
    if (resmc13.availability.value!=0){
    dequeue(fmsqueue10);
    queue (quemc13);
    seize (resmc13);
    attArrivalTime=time;
    numpart10=1;
    obsnumpart10.record(numpart10);
    dequeue (quemc13);
    delay (1.2);
    release (resmc13);

  //Seizes the resource and so empties part from queue.
  //Calculates time in queue. Delays the part in the resource
  //for the required amount of time.


  delay (10);
  queue (quemc6);

  if (resmc6.availability.value==0){
    dequeue (quemc6);
    queue (quemc9);
    while (condition (resmc9.getAvailability()==0));
    seize (resmc9);
```

```
      dequeue (quemc9);
      delay (2.9);
      release (resmc9);}
   else {seize (resmc6);
        dequeue (quemc6);
        delay (1.9);
        release (resmc6);
   }


   //Records time in system and disposes of part
   obsTimeInSystem10.record(time-attArrivalTime);
   }
   }
      dispose();
   }
   // If the part that was created above does not pass the
   //statistical distribution test, it is disposed of immediately

}
```

Vita Auctoris

Name Lia Zannier

Place of Birth: Leamington, Ontario

Year of Birth: 1977

Education St. Anne's High School, Tecumseh, Ontario
1990-1995

University of Windsor, Windsor, Ontario
1995-1999 B.A.Sc.

University of Windsor, Windsor, Ontario
1999-2003 M.A.Sc.